COMPLETE STEP-BY-STEP BUILD GUIDE

Secure Mini Chat Application (Spring Boot + JWT)

❖ PROJECT OVERVIEW

Students will build:

- User Signup
- User Login
- JWT Authentication
- Send Messages
- View Messages
- Secure APIs

PROJECT STRUCTURE

```
chat-app
 ├ controller
 ├ service
 ├ repository
 ├ entity
 ├ security
 ├ config
 └ dto
```

you should **always go in this order**:

```
Entity → Repository → Service → Controller
```

PART 1 – PROJECT SETUP

STEP 1 — Create Spring Boot Project

Go to:

https://start.spring.io

Add Dependencies:

- Spring Web
- Spring Security
- Spring Data JPA
- MySQL Driver
- Lombok

Explanation

These libraries allow:

- API creation
- Security implementation
- Database integration
- Code simplification

PART 2 – DATABASE SETUP

STEP 2 — Create Database

```
CREATE DATABASE chat_app;
```

STEP 3 — application.properties

```
#My Sql Config
spring.datasource.url=jdbc:mysql://localhost:3306/chat_app
spring.datasource.username=root
spring.datasource.password=root

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
#JWT Config
jwt.secret=your-super-secret-key
jwt.expiration=86400000
```

Explanation

- Connects Spring Boot to MySQL
- Automatically creates tables
- Stores JWT settings

PART 3 – ENTITY CREATION

STEP 4 — Create User Entity

```
User.java
@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class User {

    @Id
    @GeneratedValue
    private Long id;

    private String username;
    private String email;
```

```
    private String password;
}
```

Explanation

- `@Entity` → maps class to database table
- `id` → primary key
- `content` → chat message text
- `senderEmail` → who sent the message
- `timestamp` → when message was sent

Entity = Database table
Each field = Table column

STEP 5 — Create Message Entity

```
@Entity
@Getter
@Setter
public class Message {

    @Id
    @GeneratedValue
    private Long id;

    private String content;

    private String senderEmail;

    private LocalDateTime timestamp;
}
```

Explanation

Stores chat messages.

PART 4 – REPOSITORY LAYER

STEP 6 — UserRepository

```
@Repository
public interface UserRepository
        extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);
}
```

Explanation

Allows database operations automatically.

STEP 7 — MessageRepository

```
@Repository
public interface MessageRepository
        extends JpaRepository<Message, Long> {
}
```

PART 5 – PASSWORD ENCRYPTION

STEP 8 — PasswordConfig

```
@Configuration
public class PasswordConfig {

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

Explanation

Encrypts passwords before saving.

PART 6 – JWT IMPLEMENTATION

STEP 9 — JwtUtil

```
@Component
public class JwtUtil {

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

    @Value("${jwt.expiration}")
    private long expiration;

    public String generateToken(String email) {

        return Jwts.builder()
                .setSubject(email)
                .setIssuedAt(new Date())
                .setExpiration(new
Date(System.currentTimeMillis()+expiration))
                .signWith(Keys.hmacShaKeyFor(secret.getBytes()))
                .compact();
    }
```

```
    public String extractEmail(String token) {

        return Jwts.parserBuilder()
                .setSigningKey(secret.getBytes())
                .build()
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
}
```

Explanation

Generates and reads JWT tokens.

PART 7 – USER DETAILS SERVICE

STEP 10 — CustomUserDetailsService

```
@Service
public class CustomUserDetailsService
        implements UserDetailsService {

    @Autowired
    private UserRepository repo;

    @Override
    public UserDetails loadUserByUsername(String email) {

        User user = repo.findByEmail(email)
                .orElseThrow();

        return new org.springframework.security.core.userdetails.User(
                user.getEmail(),
                user.getPassword(),
                new ArrayList<>()
        );
    }
}
```

Explanation

Spring Security loads user using this service.

PART 8 – JWT FILTER

STEP 11 — JwtFilter

```
@Component
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
```

```
    private JwtUtil jwtUtil;

    @Autowired
    private CustomUserDetailsService userDetailsService;

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

        String header = request.getHeader("Authorization");

        if(header != null && header.startsWith("Bearer ")){

            String token = header.substring(7);
            String email = jwtUtil.extractEmail(token);

            UserDetails userDetails =
                    userDetailsService.loadUserByUsername(email);

            UsernamePasswordAuthenticationToken auth =
                    new UsernamePasswordAuthenticationToken(

userDetails,null,userDetails.getAuthorities());

            SecurityContextHolder.getContext().setAuthentication(auth);
        }

        chain.doFilter(request,response);
    }
}
```

Explanation

Intercepts every request and validates token.

PART 9 – SECURITY CONFIG

STEP 12 — SecurityConfig

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private JwtFilter jwtFilter;

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
            throws Exception {

        http.csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
```

```
                        .requestMatchers("/auth/**").permitAll()
                        .anyRequest().authenticated()
                )
                .addFilterBefore(jwtFilter,
                        UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```

Explanation

Defines which APIs are secured.

PART 10 – AUTHENTICATION SERVICE

STEP 13 — AuthService

```
@Service
public class AuthService {

    @Autowired
    private UserRepository repo;

    @Autowired
    private PasswordEncoder encoder;

    @Autowired
    private JwtUtil jwtUtil;

    public User signup(User user){
        user.setPassword(encoder.encode(user.getPassword()));
        return repo.save(user);
    }

    public String login(String email,String password){

        User user = repo.findByEmail(email).orElseThrow();

        if(!encoder.matches(password,user.getPassword())){
            throw new RuntimeException("Invalid credentials");
        }

        return jwtUtil.generateToken(email);
    }
}
```

PART 11 – CONTROLLERS

STEP 14 — AuthController

```
@RestController
@RequestMapping("/auth")
public class AuthController {
```

```
    @Autowired
    private AuthService service;

    @PostMapping("/signup")
    public User signup(@RequestBody User user){
        return service.signup(user);
    }

    @PostMapping("/login")
    public Map<String,String> login(@RequestBody User user){

        String token =
                service.login(user.getEmail(),user.getPassword());

        return Map.of("token",token);
    }
}
```

STEP 15 — Message Service

MessageService.java

```
@Service
public class MessageService {

    @Autowired
    private MessageRepository messageRepository;

    public Message saveMessage(String content, String senderEmail) {

        Message message = new Message();
        message.setContent(content);
        message.setSenderEmail(senderEmail);
        message.setTimestamp(LocalDateTime.now());

        return messageRepository.save(message);
    }

    public List<Message> getAllMessages() {
        return messageRepository.findAll();
    }
}
```

Explanation

- **Business logic lives here**
- Controller should not create objects
- Service:
  - creates Message object
  - sets data
  - saves to DB

Service = Business rules + logic

🔹 **Message Controller**

📄 **MessageController.java**

```
@RestController
@RequestMapping("/messages")
public class MessageController {

    @Autowired
    private MessageService messageService;

    @PostMapping
    public Message sendMessage(@RequestBody Map<String, String> request)
{

        String content = request.get("content");

        Authentication auth =
                SecurityContextHolder.getContext().getAuthentication();

        String senderEmail = auth.getName();

        return messageService.saveMessage(content, senderEmail);
    }

    @GetMapping
    public List<Message> getMessages() {
        return messageService.getAllMessages();
    }
}
```

Explanation

- Controller:
  - accepts HTTP request
  - extracts input
  - calls service
- It **does NOT talk to DB directly**

Controller = HTTP layer only

Security Filter Chain.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{

    http
            .cors(Customizer.withDefaults())    // ⭐ Enables CORS
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers("/auth/**").permitAll()
                    .anyRequest().authenticated()
            )
            .addFilterBefore(jwtFilter,
```

```
UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

PART 12 – TESTING

Signup API

```
POST /auth/signup
```

Login API

```
POST /auth/login
```

Send Message API

```
Authorization: Bearer TOKEN
```

PART 13 – FRONTEND POLLING + FRONTEND INTEGRATION

Auto Load Messages

```
setInterval(loadMessages, 5000);
```

Explanation

Calls API every 5 seconds.

FINAL PROJECT FLOW

```
User Signup
   ↓
User Login
   ↓
JWT Generated
   ↓
Token Stored in Frontend
   ↓
JWT Filter Validates Requests
   ↓
Secure Messaging
```

OUTCOME

You will understand:

- Spring Security
- JWT Authentication
- REST APIs
- Database integration
- Full-stack communication