Resolução Lista 1 - Teórica

1) (Fácil) O que é SOLID?
Resultado esperado:
SOLID é um acrônimo que representa cinco princípios de programação, teorizada por Robert C. Martin no início dos anos 2000.

2) (Fácil) Quais são os princípios do SOLID?
Resultado esperado:
S = Single Responsability Principle (Princípio da Responsabilidade Única)
O = Open/Closed Principle (Princípio do Aberto/Fechado)
L = Liskov Substitution Principle (Princípio da Substituição de Liskov)
I = Interface Segregation Principle (Princípio da Segregação de Interfaces)
D = Dependency Inversion Principle (Princípio da Inversão de Dependência)

3) (Fácil) Defina o princípio da responsabilidade única
Resultado esperado:
Uma classe deve ter um e somente um motivo para mudar. Uma classe deve ser muito especializada, tendo apenas uma única tarefa.

4) (Fácil) Defina o princípio do Aberto/Fechado
Resultado esperado:
Objetou ou entidades devem estar abertos para extensão, mas fechados para modificação

5) (Fácil) Defina o Princípio da Substituição de Liskov
Resultado esperado:
Uma classe derivada deve ser substituível por sua classe base.

6) (Médio) Defina o Princípio da Segregação de Interface
Resultado esperado:
Uma classe não deve ser forçada a implementar interfaces e métodos que não serão utilizados. É melhor criar interfaces mais especificas do que criar uma interface enorme.

7) (Médio) Defina o Princípio de Inversão de Dependência
Resultado esperado:
Devemos depender sempre de abstrações e não de implementações. Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender da abstração.

8) (Médio) O que são testes unitários e qual é a sua importância no ciclo de desenvolvimento de Software moderno?
Resultado esperado:
São testes automatizados que testam nosso software em sua menor fração, em outras palavras, efetua testes nos métodos das classes implementadas. Testes unitários são importantes pois são muito “baratos” e previnem uma quantidade considerável de bugs no sistema.

9) (Médio) Considerando a classe “Pessoa” apresentada abaixo, implemente uma classe de teste, com pelo menos dois casos de teste para o método “calcularIdade”

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Pessoa {
    private Integer id;
    private String nome;
    private String sobrenome;
    private LocalDate dataNascimento;
    public Long calculaIdade(LocalDate dataDeHoje) {
        if (dataNascimento == null) {
            return 0L;
        }
        return ChronoUnit.YEARS.between(dataNascimento, dataDeHoje);
    }
}
```

Resultado esperado:

```
@Test
public void testIdadeMaiorQueZero() {
    Pessoa p = new Pessoa();
    p.setDataNascimento(LocalDate.of(2017, 02,02));
    Long idade = p.calculaIdade(LocalDate.of(2023,02,23));
    Assertions.assertEquals(6, idade);
}
@Test
public void testIdadeDeBebes() {
    Pessoa p = new Pessoa();
    p.setDataNascimento(LocalDate.of(2023, 02,02));
    Long idade = p.calculaIdade(LocalDate.of(2023,02,23));
    Assertions.assertEquals(0, idade);
}
```

10) (Médio) Dado o método de configuração “authenticationManager” apresentado abaixo, apresente quais são as exigências necessárias para que o usuarioService e usuário sejam consideradas como válidas para o Spring Security

```
@Bean
public AuthenticationManager authenticationManager(HttpSecurity http,
BCryptPasswordEncoder bCryptPasswordEncoder,
      UsuarioService usuarioService) throws Exception {

   if (!usuarioService.userExists("admin")) {
      usuarioService.createAdminUser();
   }
   return http
         .getSharedObject(AuthenticationManagerBuilder.class)
         .userDetailsService(usuarioService)
         .passwordEncoder(passwordEncoder())
         .and().build();
}
```

Resultado esperado:

Para que o Spring Security consiga carregar nossas informações de usuário, é preciso que a classe de serviço implemente a interface “UserDetailsService” e a classe de entidade implemente a interface “UserDetails”, conforme exemplo abaixo:

```
public class UsuarioService implements UserDetailsService { }

public class Usuario implements UserDetails { }
```

11) (Difícil) Dado o método de configuração filterChain do Spring Secutity, altere o código de forma que seja possível acessar também a url ***/sobre,*** sem a necessidade de autenticação.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
   http
      .csrf().disable()
      .authorizeRequests()
         .antMatchers("/login").permitAll()
         .antMatchers("/swagger**").permitAll()
      .anyRequest().authenticated()
      .and().exceptionHandling()
         .authenticationEntryPoint(jwtAuthenticationEntryPoint)
      .and().sessionManagement()
         .sessionCreationPolicy(SessionCreationPolicy.STATELESS);

   http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFil-
ter.class);

   return http.build();
}
```

Resultado esperado:

```
.antMatchers("/login").permitAll()
.antMatchers("/swagger**").permitAll()
.antMatchers("/sobre**").permitAll()
```

12) (Difícil) Considerando a classe de testes apresentada abaixo, explique qual é a diferença entre **@InjectMock** e **@Mock**

```
@ExtendWith(MockitoExtension.class)
public class ProdutoServiceTest {

    @InjectMocks
    private ProdutoService produtoService;

    @Mock
    private ProdutoRepository produtoRepository;

    @Test
    public void listarTudoComDoisRegistrosDeveRetornarCountDois() {
        // Arrange
        List<Produto> listaFake = criaListaFake();
        when(produtoRepository.findAll()).thenReturn(listaFake);

        // Act
        int quantidade = produtoService.listarTudo().size();

        // Assert
        Assertions.assertEquals(2, quantidade);
    }
    private List<Produto> criaListaFake() {
        List<Produto> lista = new ArrayList<>();
        lista.add(new Produto());
        lista.add(new Produto());
        return lista;
    }
}
```

Resultado esperado:

A anotação @Mock indica que o objeto anotado é passível de ser modificado pelo Mockito,

enquanto a anotação @InjectMock indica que o objeto anota será monitorado para injetar os mocks dentro dele.

13) (Difícil) Considerando a classe JwtRequestFilter apresentada abaixo, indique em que momento do código o usuário é autenticado e considerado válido dentro do Spring Securitty

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

   @Autowired
   private UsuarioService usuarioService;

   @Autowired
   private JwtTokenService jwtTokenService;

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

      final String requestTokenHeader = request.getHeader("Authorization");

      String username = null;
      String jwtToken = null;

      // JWT Token está no form "Bearer token". Remova a palavra Bearer e pegue
      // somente o Token
      if (requestTokenHeader != null && requestTokenHeader.startsWith("Bearer
")) {
         jwtToken = requestTokenHeader.substring(7);
         try {
            username = jwtTokenService.getUsernameFromToken(jwtToken);
         } catch (IllegalArgumentException e) {
            System.out.println("Unable to get JWT Token");
         } catch (ExpiredJwtException e) {
            System.out.println("JWT Token has expired");
         }
      } else {
         logger.warn("JWT Token does not begin with Bearer String");
      }
      // Tendo o token, o validamos.
      if (username != null && SecurityContextHold-
er.getContext().getAuthentication() == null) {
         UserDetails userDetails =
this.usuarioService.loadUserByUsername(username);

         if (jwtTokenService.validateToken(jwtToken, userDetails)) {
            UsernamePasswordAuthenticationToken usernamePasswordAuthentica-
tionToken = new UsernamePasswordAuthenticationToken(
                  userDetails, null, userDetails.getAuthorities());
            usernamePasswordAuthenticationToken
                  .setDetails(new WebAuthenticationDetails-
Source().buildDetails(request));

            SecurityContextHold-
er.getContext().setAuthentication(usernamePasswordAuthenticationToken);
         }

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

```
    }
}
```

Resultado esperado:
No momento de execução do comando SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken) é adicionado dentro da sessão do SpringSecurity as informações do usuário logado. A partir das próximas chamadas e verificações, o usuário é considerado válido.

14) (Difícil) Explique, com suas palavras, o processo de autenticação da API utilizando Token JWT.
Resultado esperado:
O processo é divido em duas etapas:
A autenticação é realizada através de um POST na url **/login,** informando os dados de usuário e senha. Se o usuário for válido, será gerado um Token JWT e retornado para a aplicação cliente.
Com o token em mãos, o mesmo deve ser repassado no header de todas as requisições, no campo Authorization. Antes de executar o código de cada controller, será aplicado um filtro que valida o conteúdo do JWT e decide se ele está válido ou não.

15) (Difícil) Dado uma aplicação Spring Boot, quais são os procedimentos necessários para habilitar as funcionalidades de Segurança e autenticação por JWT
Resultado esperado:
Instalação das dependências no POM.XML
Implementação das classes que irão representar o Usuário e seus serviços
Implementação das classes de configuração das Rotas e forma de autenticação
Implementação das classes de filtro para validação do JWT
Implementação das classes de controller para efetuar o login dos usuários