@Component  
@Slf4j  
**public class** SecurityFilter **extends** OncePerRequestFilter **implements** Filter {  
  
  
 **private static final** String ***AZURE\_HEADER\_CLIENT\_CERT*** = **"X-ARR-ClientCert"**;  
 **private static final** String ***BEGIN\_CERT*** = **"-----BEGIN CERTIFICATE-----\n"**;  
 **private static final** String ***END\_CERT*** = **"\n-----END CERTIFICATE-----"**;  
 **private static final** String ***ORG*** = **"ABN AMRO Bank N.V."**;  
  
 **private static final** List<String> ***EXCLUDE\_URL*** = Arrays.*asList*(***HEALTH\_CHECK***,  
 ***ACTUATOR\_HEALTH\_CHECK***,**"/context-path/v3/api-docs"**);  
  
 @Autowired  
 **private** Map<String,String> **keyvaultVariables**;  
  
 **public** SecurityFilter(Map<String, String> keyvaultVariables) {  
 **this**.**keyvaultVariables** = keyvaultVariables;  
 }  
  
 */\*\*  
 \* All security validation should happen here  
 \*/* @Bean  
 **public** FilterRegistrationBean<SecurityFilter> logFilter() {  
 FilterRegistrationBean<SecurityFilter> registrationBean = **new** FilterRegistrationBean<>();  
 registrationBean.setFilter(**this**);  
 registrationBean.addUrlPatterns(**"/rest/api"**+***DOCUMENT\_V1\_SEARCH***,**"/rest/api"**+***DOCUMENT\_V1\_SEARCH\_CONCORD***);  
 **return** registrationBean;  
 }  
  
 @Override  
 **protected void** doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, FilterChain filterChain) **throws** ServletException, IOException {  
 {  
  
 **if** (httpServletRequest.getServletPath().equalsIgnoreCase(**"/rest/api"**+***DOCUMENT\_V1\_SEARCH***)) {  
 **if** ((StringUtils.*isNotBlank*(httpServletRequest.getHeader(***AZURE\_HEADER\_CLIENT\_CERT***))  
 && (StringUtils.*isNotBlank*(httpServletRequest.getHeader(***MDC\_CONSUMER\_ID***))))) {  
 String certificate = httpServletRequest.getHeader(***AZURE\_HEADER\_CLIENT\_CERT***);  
 certificate = certificate.replaceAll(***BEGIN\_CERT***, **""**).replaceAll(***END\_CERT***, **""**);  
 **try** {  
 CertificateFactory cf = CertificateFactory.*getInstance*(**"X.509"**);  
 X509Certificate x509Certificate =  
 (X509Certificate) cf.generateCertificate(**new** ByteArrayInputStream(Base64.*getDecoder*().decode(certificate)));  
  
 **if** (validateClientCertificate(x509Certificate)) {  
 filterChain.doFilter(httpServletRequest, httpServletResponse);  
 } **else** {  
 handleException(httpServletResponse, ExceptionCode.***CLIENT\_CERTIFICATE\_INVALID***.name(),  
 ExceptionCode.***CLIENT\_CERTIFICATE\_INVALID***.getMessage(),  
 ObjectUtils.*isEmpty*(httpServletRequest.getHeader(***HEADER\_TRACE\_ID***)) ? UUID.*randomUUID*().toString(): httpServletRequest.getHeader(***HEADER\_TRACE\_ID***),  
 **"Consumer: "**+httpServletRequest.getHeader(***HEADER\_CONSUMER\_ID***)+**" with Cert:"**+ httpServletRequest.getHeader(***AZURE\_HEADER\_CLIENT\_CERT***));  
 }  
  
 } **catch** (CertificateException | IOException e) {  
 ***log***.error(**"Exception occured while validating client certificate for incoming requests "**, e);  
 }  
  
 } **else** {  
 handleException(httpServletResponse,  
 ExceptionCode.***CONSUMER\_ID\_MISSING***.name(),  
 ExceptionCode.***CONSUMER\_ID\_MISSING***.getMessage(),  
 ObjectUtils.*isEmpty*(httpServletRequest.getHeader(***HEADER\_TRACE\_ID***)) ? UUID.*randomUUID*().toString(): httpServletRequest.getHeader(***HEADER\_TRACE\_ID***),  
 **"Consumer: "**+httpServletRequest.getHeader(***HEADER\_CONSUMER\_ID***)+**" with Cert:"**+ httpServletRequest.getHeader(***AZURE\_HEADER\_CLIENT\_CERT***));  
 }  
 }  
 **else if** (httpServletRequest.getServletPath().equalsIgnoreCase(**"/rest/api"**+***DOCUMENT\_V1\_SEARCH\_CONCORD***)) {  
 **if** (StringUtils.*isNotBlank*(httpServletRequest.getHeader(***MDC\_AUTHORIZATION***))) {  
 filterChain.doFilter(httpServletRequest,httpServletResponse);  
 } **else** {  
 handleException(httpServletResponse,  
 ExceptionCode.***ACCESS\_DENIED\_CONCORD***.name(),  
 ExceptionCode.***ACCESS\_DENIED\_CONCORD***.getMessage(),  
 ObjectUtils.*isEmpty*(httpServletRequest.getHeader(***HEADER\_TRACE\_ID***)) ? UUID.*randomUUID*().toString(): httpServletRequest.getHeader(***HEADER\_TRACE\_ID***),  
 ***HEADER\_AUTHORIZATION***);  
 }  
 }  
 }  
 }  
 */\*\*  
 \*  
 \** ***@param httpServletResponse*** *\** ***@param name*** *\** ***@param message*** *\** ***@throws*** *IOException  
 \*/* **public void** handleException(HttpServletResponse httpServletResponse, String name, String message, String traceId, String header) **throws** IOException {  
 httpServletResponse.setHeader(HttpHeaders.***CONTENT\_TYPE***, **"application/json"**);  
 httpServletResponse.setStatus(HttpStatus.***BAD\_REQUEST***.value());  
 httpServletResponse.setContentType(**"application/json"**);  
 ObjectMapper objectMapper = **new** ObjectMapper();  
 Error error=**new** Error(name, message, HttpStatus.***BAD\_REQUEST***.value(),traceId);  
 ErrorList errorList = **new** ErrorList();  
 errorList.getErrors().add(error);  
 ***log***.info(header);  
 httpServletResponse.getWriter().write(objectMapper.writeValueAsString(errorList));  
 }  
  
  
 */\*\*  
 \* Validates client certificate in terms on date to check that certificate is not expired, and organization to check  
 \* OU and O  
 \*  
 \** ***@param certificate*** *x509 certificate received in the request header  
 \** ***@return*** *true if the certificate is valid  
 \*/* **public boolean** validateClientCertificate(X509Certificate certificate) {  
 **boolean** isValid = **true**;  
 isValid = certificateHasNotExpired(certificate);  
 String name = certificate.getSubjectX500Principal().getName();  
 **if** (!name.contains(**"OU="** + **keyvaultVariables**.get(**"TOKEN-CN"**)) || !name.contains(**"O="** + ***ORG***)) {  
 isValid = **false**;  
 }  
  
 **return** isValid;  
 }  
  
 */\*\*  
 \* Check certificate's timestamp.  
 \*  
 \** ***@return*** *Returns true if the certificate has not expired. Returns false if it has expired.  
 \*/* **private boolean** certificateHasNotExpired(X509Certificate certificate) {  
 Date currentTime = **new** Date();  
 **try** {  
 certificate.checkValidity(currentTime);  
 } **catch** (CertificateExpiredException e) {  
 ***log***.error(**"Client certificate is expired "** + e);  
 **return false**;  
 } **catch** (CertificateNotYetValidException e) {  
 ***log***.error(**"Client certificate is not yet valid "** + e);  
 **return false**;  
 }  
 **return true**;  
 }  
}

…………………

**private boolean** checkTypes(String documentName, String serachText)  
{  
  
 List<String> serachTextarray= Arrays.*asList*(serachText.split(**"^\\s\*$"**)) ;  
 Pattern pattern = Pattern.*compile*(documentName);  
 **for**(**int** i=0;i< serachTextarray.size();i++)  
 {  
 StringBuilder builder=**new** StringBuilder();  
 builder.append(**"(?i)(.\*)"**).append(serachTextarray.get(i)).append(**"(.\*)"**);  
 pattern= returnPattern(Pattern.*compile*(builder.toString()));  
 }  
 **return** pattern.matcher(documentName).find();  
  
}

…………………….

docSourceList = (Stream.*concat*(drmDocSourceTypeList.stream(), eDocSourceList.stream()).collect(*toList*())).stream().collect(  
 Collectors.*collectingAndThen*(  
 Collectors.*groupingBy*(DrmDocumentSourceAggregation::getSourceName,Collectors.*summingLong*(DrmDocumentSourceAggregation::getCount)),  
 map -> map.entrySet().stream()  
 .map(e -> **new** DrmDocumentSourceAggregation(e.getKey(), e.getValue()))  
 .collect(Collectors.*toList*())));

……………

Map<DocumentType, Long> edocumentType =documentList.stream().collect(Collectors.*groupingBy*(p -> p.getDocumentType(),  
 Collectors.*counting*()));  
List<DrmDocumentAggregation> eDoctypeList= edocumentType.entrySet().stream().map(e->**new** DrmDocumentAggregation(e.getKey(),e.getValue())).collect(Collectors.*toList*());  
**……………**

**package** com.abnamro.arss.document.search.config;  
  
  
**import** com.abnamro.arss.document.search.common.exception.InternalServerException;  
**import** com.abnamro.arss.document.search.common.util.DecryptionUtil;  
**import** com.azure.identity.DefaultAzureCredentialBuilder;  
**import** com.azure.security.keyvault.certificates.CertificateClient;  
**import** com.azure.security.keyvault.certificates.CertificateClientBuilder;  
**import** com.azure.security.keyvault.certificates.models.KeyVaultCertificate;  
**import** com.azure.security.keyvault.certificates.models.KeyVaultCertificateWithPolicy;  
**import** com.azure.security.keyvault.secrets.SecretClient;  
**import** com.azure.security.keyvault.secrets.models.KeyVaultSecret;  
**import** io.netty.handler.ssl.SslContext;  
**import** io.netty.handler.ssl.SslContextBuilder;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.beans.factory.annotation.Value;  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.DependsOn;  
**import** org.springframework.context.annotation.Profile;  
**import** org.springframework.core.io.Resource;  
**import** org.springframework.http.client.reactive.ReactorClientHttpConnector;  
**import** org.springframework.stereotype.Component;  
**import** org.springframework.web.reactive.function.client.ExchangeStrategies;  
**import** org.springframework.web.reactive.function.client.WebClient;  
**import** reactor.netty.http.client.HttpClient;  
  
**import** javax.net.ssl.KeyManagerFactory;  
**import** javax.net.ssl.SSLException;  
**import** javax.net.ssl.TrustManagerFactory;  
**import** java.io.ByteArrayInputStream;  
**import** java.io.IOException;  
  
**import** java.security.\*;  
**import** java.security.cert.Certificate;  
**import** java.security.cert.CertificateException;  
**import** java.security.cert.CertificateFactory;  
**import** java.util.Base64;  
**import** java.util.Map;  
  
  
@Component  
**public class** ApplicationConfig {  
 @Value(**"${search.document.base\_url}"**)  
 **private** String **base\_url**;  
 @Value(**"${elastic.trustStoreCrypto}"**)  
 **private** String **trustStoreCrypto**;  
 @Value(**"${azure.key.vault.uri}"**)  
 **private** String **keyVaultUrl**;  
 **private final** SecretClient **secretClient**;  
 @Value(**"${base.trust}"**)  
 **private** Resource **trustStore**;  
 @Autowired  
 **private** Map<String,String> **keyvaultVariables**;  
  
  
 **public** ApplicationConfig(SecretClient secretClient, Map<String, String> keyvaultVariables) {  
 **this**.**secretClient** = secretClient;  
 **this**.**keyvaultVariables** = keyvaultVariables;  
 }  
  
 @Bean  
 @DependsOn({**"keyvaultVariables"**})  
 **public** WebClient webClientTLSMA() **throws** IOException, CertificateException{  
 **try** {  
  
 KeyStore truststore = KeyStore.*getInstance*(**"JKS"**);  
 truststore.load(**trustStore**.getInputStream(), DecryptionUtil.*decryptCrypto*(**keyvaultVariables**.get(**trustStoreCrypto**)).toCharArray());  
 KeyManagerFactory keyManagerFactory = KeyManagerFactory.*getInstance*(KeyManagerFactory.*getDefaultAlgorithm*());  
 keyManagerFactory.init(generateCertificate(**"cyberark"**),DecryptionUtil.*decryptCrypto*(**keyvaultVariables**.get(**trustStoreCrypto**)).toCharArray());  
 TrustManagerFactory trustManagerFactory = TrustManagerFactory.*getInstance*(TrustManagerFactory.*getDefaultAlgorithm*());  
 trustManagerFactory.init(truststore);  
 **final int** size = 16 \* 1024 \* 1024;  
 **final** ExchangeStrategies strategies = ExchangeStrategies.*builder*()  
 .codecs(codecs -> codecs.defaultCodecs().maxInMemorySize(size))  
 .build();  
 HttpClient httpClient = HttpClient.*create*()  
 .secure(sslSpec -> {  
 **try** {  
 sslSpec.sslContext(SslContextBuilder.*forClient*()  
 .keyManager(keyManagerFactory)  
 .trustManager(trustManagerFactory)  
 .build());  
 } **catch** (SSLException e) {  
 **throw new** InternalServerException(**"Technical issue within the Audit Service"**);  
 }  
 });  
 **return** WebClient.*builder*().exchangeStrategies(strategies)  
 .clientConnector(**new** ReactorClientHttpConnector(httpClient))  
 .baseUrl(**base\_url**).build();  
 }  
 **catch** (KeyStoreException | UnrecoverableKeyException | NoSuchAlgorithmException e) {  
 **throw new** InternalServerException(**"Technical issue within the Service"**);  
  
 }  
 }  
  
  
 */\*\*  
 \* Method to generate keystore  
 \** ***@param keyStream*** *\** ***@param certificate*** *\** ***@param drpbCertName*** *\** ***@return*** *\** ***@throws*** *KeyStoreException  
 \** ***@throws*** *CertificateException  
 \** ***@throws*** *NoSuchAlgorithmException  
 \** ***@throws*** *IOException  
 \** ***@throws*** *UnrecoverableKeyException  
 \*/* **private** KeyStore getKeystore(ByteArrayInputStream keyStream, Certificate certificate, String drpbCertName)  
 **throws** KeyStoreException, CertificateException, NoSuchAlgorithmException, IOException,  
 UnrecoverableKeyException {  
 KeyStore rsaKeyGenerator = KeyStore.*getInstance*(**"JKS"**);  
 rsaKeyGenerator.load(keyStream, **null**);  
 Key rsaPrivateKey = rsaKeyGenerator.getKey(rsaKeyGenerator.aliases().nextElement(),  
 **""**.toCharArray());  
 KeyStore keyStore = KeyStore.*getInstance*(KeyStore.*getDefaultType*());  
 keyStore.load(**null**);  
  
 keyStore.setKeyEntry(**"mycert"**, rsaPrivateKey, DecryptionUtil.*decryptCrypto*(**keyvaultVariables**.get(**trustStoreCrypto**)).toCharArray(),  
 **new** Certificate[] { certificate });  
 **return** keyStore;  
 }  
  
 */\*\*  
 \* This method to generate certificate  
 \** ***@param certName*** *Name of the certificate  
 \** ***@return*** *keystore file to be returned  
 \*/* **public** KeyStore generateCertificate(String certName)  
 {  
 KeyStore keyStore =**null**;  
 **try** {  
 CertificateClient certificateClient = **new** CertificateClientBuilder()  
 .vaultUrl(**keyVaultUrl**)  
 .credential(**new** DefaultAzureCredentialBuilder().build())  
 .buildClient();  
 KeyVaultCertificate cybertrust = certificateClient.getCertificate(certName);  
 KeyVaultCertificateWithPolicy cybertrustPolicy = certificateClient.getCertificate(certName);  
 CertificateFactory cf = CertificateFactory.*getInstance*(**"X.509"**);  
 Certificate certificate = cf.generateCertificate(**new** ByteArrayInputStream(cybertrust.getCer()));  
 KeyVaultSecret secret = **secretClient**.getSecret(certName,  
 cybertrustPolicy.getProperties().getVersion());  
 String base64PrivateKey = secret.getValue();  
 **byte**[] rawPrivateKey = Base64.*getDecoder*().decode(base64PrivateKey);  
 keyStore = getKeystore(**new** ByteArrayInputStream(rawPrivateKey), certificate, certName);  
  
 }  
 **catch** (KeyStoreException | IOException | UnrecoverableKeyException | NoSuchAlgorithmException | CertificateException e) {  
 **throw new** InternalServerException(**"Technical issue within the Service"**);  
  
 }  
 **return** keyStore;  
 }  
}  
……

**package** com.abnamro.arss.document.search.common.cyberark;  
  
**import** com.abnamro.arss.document.search.common.util.DecryptionUtil;  
**import** com.abnamro.arss.document.search.config.ApplicationConfig;  
**import** com.abnamro.arss.document.search.exception.CyberArkConnectionException;  
**import** org.apache.hc.client5.http.impl.classic.CloseableHttpClient;  
**import** org.apache.hc.client5.http.impl.classic.HttpClients;  
**import** org.apache.hc.client5.http.impl.io.PoolingHttpClientConnectionManagerBuilder;  
**import** org.apache.hc.client5.http.io.HttpClientConnectionManager;  
**import** org.apache.hc.client5.http.ssl.SSLConnectionSocketFactory;  
**import** org.apache.http.ssl.PrivateKeyDetails;  
**import** org.apache.http.ssl.PrivateKeyStrategy;  
**import** org.apache.http.ssl.SSLContextBuilder;  
**import** org.apache.http.ssl.TrustStrategy;  
**import** org.json.JSONObject;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.beans.factory.annotation.Value;  
**import** org.springframework.boot.web.client.RestTemplateBuilder;  
**import** org.springframework.http.ResponseEntity;  
**import** org.springframework.http.client.ClientHttpRequestFactory;  
**import** org.springframework.http.client.HttpComponentsClientHttpRequestFactory;  
**import** org.springframework.stereotype.Component;  
**import** org.springframework.web.client.RestTemplate;  
  
**import** javax.net.ssl.KeyManagerFactory;  
**import** java.io.IOException;  
**import** java.net.Socket;  
**import** java.net.URLEncoder;  
**import** java.security.\*;  
**import** java.security.cert.CertificateException;  
**import** java.security.cert.X509Certificate;  
**import** java.util.Map;  
*/\*  
Class to create cyberarkclient for password retrival  
 \*/*@Component  
**public class** CyberArkClient {  
 **private static final** String ***UTF\_ENCODE*** = **"UTF-8"**;  
 **private static final** String ***CONTENT*** = **"Content"**;  
 **private static final** String ***ERROR\_CONNECTING\_CYBERARK*** = **"Connection error while retrieving password from CyberArk: "**;  
 **private static final** String ***NEXT*** = **"?"**;  
  
 @Value(**"${cyberark.app.id}"**)  
 **private** String **appId**;  
  
 @Value(**"${cyberark.safe.id}"**)  
 **private** String **safeId**;  
  
 @Value(**"${cyberark.rest.url}"**)  
 **private** String **restURL**;  
  
  
 @Value(**"${es.cyberark.object}"**)  
 **private** String **cyberarkObject**;  
 @Autowired  
 RestTemplateBuilder **restTemplateBuilder**;  
 @Value(**"${elastic.trustStoreCrypto}"**)  
 **private** String **trustStoreCrypto**;  
 @Autowired  
 **private** Map<String,String> **keyvaultVariables**;  
 @Autowired  
 **private** ApplicationConfig **applicationConfig**;  
  
 */\*  
 Constructor to initilize the keyvault variables  
 \*/* **public** CyberArkClient(Map<String, String> keyvaultVariables) {  
 **this**.**keyvaultVariables** = keyvaultVariables;  
 }  
  
 */\*\*  
 \* Method to get secret from CyberArk based on parameter, secrets can be retrieved based on user name or object string  
 \* in CyberArk, if user name is unique in CyberArk then object parameter will be ignored, otherwise the secret will be  
 \* retrieved based on object value, since object is always unique in CyberArk and in this case user name should be  
 \* passed as null  
 \*  
 \** ***@param objectId*** *object string in CyberArk  
 \** ***@param appId*** *object string in CyberArk  
 \** ***@param safeId*** *object string in CyberArk  
 \** ***@return*** *secret  
 \*/* **public** String getPassword(String objectId, String appId, String safeId) {  
 **try** {  
 appId = URLEncoder.*encode*(appId, ***UTF\_ENCODE***);  
 safeId = URLEncoder.*encode*(safeId, ***UTF\_ENCODE***);  
 objectId = URLEncoder.*encode*(objectId, ***UTF\_ENCODE***);  
 String parameterizedUrl = getParameterizedUrl(**restURL**, appId, safeId, objectId);  
  
 ResponseEntity<String> response = getRestTemplate().getForEntity(parameterizedUrl, String.**class**);  
  
 String responseString = response.getBody();  
 JSONObject jsonObject = **new** JSONObject(responseString);  
  
 **return** jsonObject.getString(***CONTENT***);  
 } **catch** (IOException e) {  
 **throw new** CyberArkConnectionException(***ERROR\_CONNECTING\_CYBERARK*** + e);  
 }  
 }  
  
 */\*\*  
 \* Get rest template  
 \*  
 \** ***@return*** *custom RestTemplate  
 \*/* **public** RestTemplate getRestTemplate() {  
 **return restTemplateBuilder**.requestFactory(() -> validateSSL()).build();  
 }  
  
 */\*\*  
 \* This method sends valid certificate and makes SSL connection  
 \*  
 \** ***@return*** *Trusted http client  
 \*/* **public** HttpComponentsClientHttpRequestFactory validateSSL() {  
 SSLConnectionSocketFactory socketFactory = **null**;  
 ClientHttpRequestFactory requestFactory = **null**;  
 **try** {  
 KeyManagerFactory keyManagerFactory = KeyManagerFactory.*getInstance*(KeyManagerFactory.*getDefaultAlgorithm*());  
 KeyStore ks = **applicationConfig**.generateCertificate(**"cyberark"**);  
 keyManagerFactory.init(ks, DecryptionUtil.*decryptCrypto*(**keyvaultVariables**.get(**trustStoreCrypto**)).toCharArray());  
 socketFactory = **new** SSLConnectionSocketFactory(**new** SSLContextBuilder().loadTrustMaterial(**new** TrustStrategy() {  
 @Override  
 **public boolean** isTrusted(X509Certificate[] chain, String authType) **throws** CertificateException {  
 **return true**;  
 }  
 }).loadKeyMaterial(ks, DecryptionUtil.*decryptCrypto*(**keyvaultVariables**.get(**trustStoreCrypto**)).toCharArray(), **new** PrivateKeyStrategy() {  
 @Override  
 **public** String chooseAlias(Map<String, PrivateKeyDetails> map, Socket socket) {  
 **return "mycert"**;  
 }  
 }).build());  
  
 HttpClientConnectionManager connectionManager = PoolingHttpClientConnectionManagerBuilder.*create*().setSSLSocketFactory(socketFactory).build();  
 **final** CloseableHttpClient httpClient = HttpClients.*custom*().setConnectionManager(connectionManager).build();  
 requestFactory = **new** HttpComponentsClientHttpRequestFactory(httpClient);  
 } **catch** (KeyManagementException | NoSuchAlgorithmException | KeyStoreException | UnrecoverableKeyException e) {  
  
 **throw new** IllegalStateException(**"Failed to setup client SSL context"**, e);  
 }  
  
  
 **return** (HttpComponentsClientHttpRequestFactory) requestFactory;  
 }  
  
 */\*\*  
 \* Method to build url with parameters  
 \*  
 \** ***@param restURL*** *rest api of CyberArk  
 \** ***@param appId*** *unique app id in CyberArk  
 \** ***@param safeId*** *unique safe id in CyberArk  
 \** ***@param objectId*** *object string in CyberArk  
 \** ***@return*** *parameterized URL  
 \** ***@throws*** *IOException some sort of I/O exception  
 \*/* **public** String getParameterizedUrl(String restURL, String appId, String safeId, String objectId) {  
 StringBuilder sb = **new** StringBuilder();  
 sb.append(restURL);  
 sb.append(***NEXT*** + **"appId="** + appId);  
 sb.append(**"&Safe="** + safeId);  
 sb.append(**"&Object="** + objectId);  
 **return** sb.toString();  
 }  
}

……………..

**package** com.abnamro.arss.document.search.es.mapper;  
  
**import** com.abnamro.arss.document.search.common.model.DocumentStatus;  
**import** com.abnamro.arss.document.search.common.model.DrmDocument;  
**import** com.abnamro.arss.document.search.common.model.security.SecurityLevel;  
**import** com.abnamro.arss.document.search.es.model.ESDocument;  
**import** org.mapstruct.Mapper;  
**import** org.mapstruct.Mapping;  
**import** org.mapstruct.Named;  
  
**import** java.text.ParseException;  
**import** java.text.SimpleDateFormat;  
**import** java.util.Calendar;  
**import** java.util.Date;  
**import** java.util.GregorianCalendar;  
**import** java.util.Locale;  
  
*/\*\*  
 \* Maps ESDocument object to DrmDocument object  
 \*/*@Mapper(config = MappingConfig.**class**, componentModel = **"spring"**)  
**public interface** DrmDocumentESDocumentMapper {  
  
 */\*\*  
 \* Converts @link{ESDocument} to @link{DrmDocument}  
 \*  
 \** ***@param esDocument*** *model class containing metadata of elastic search Document object  
 \** ***@return*** *Object class containing the information for the model of Drm document metadata  
 \*/* @Mapping(source = **"documentId"**, target = **"documentId"**, qualifiedByName = **"constructId"**)  
 @Mapping(source = **"docStatus"**, target = **"documentStatus"**, qualifiedByName = **"toEnum"**)  
 @Mapping(source = **"retentionStartDate"**, target = **"retentionPolicyInfo.retentionStartDateTime"**, qualifiedByName = **"toCalendar"**)  
 @Mapping(source = **"retentionEndDate"**, target = **"retentionPolicyInfo.retentionEndDateTime"**, qualifiedByName = **"toCalendar"**)  
 @Mapping(source = **"retentionPolicy"**, target = **"retentionPolicyInfo.retentionPeriod"**)  
 @Mapping(source = **"scope"**, target = **"scopes"**)  
 @Mapping(source = **"documentType"**, target = **"documentType.type"**)  
 @Mapping(source = **"subType"**, target = **"documentType.subType"**)  
 @Mapping(source = **"creationDate"**, target = **"creationDate"**, qualifiedByName = **"toCalendar"**)  
 @Mapping(source = **"lastModifiedDate"**, target = **"modifyDate"**, qualifiedByName = **"toCalendar"**)  
 @Mapping(source = **"lastModifierUserId"**, target = **"modifierName"**)  
 @Mapping(source = **"securityLevel"**, target = **"securityLevel"**, qualifiedByName = **"toUppercase"**)  
 @Mapping(source = **"creatorConsumerId"**, target = **"creatorName"**)  
 @Mapping(source = **"sourceCreatorUserId"**, target = **"sourceCreatorUserId"**)  
 @Mapping(source = **"sourceCreationDateTime"**, target = **"sourceCreationDateTime"**)  
 @Mapping(source = **"description"**, target = **"description"**)  
 @Mapping(source = **"acl"**, target = **"aclLabels"**)  
 @Mapping(source = **"sourceVersionLabel"**, target = **"sourceVersionLabel"**)  
 @Mapping(source = **"sourceLabel"**, target = **"sourceLabel"**)  
 *//@Mapping(source = "sourceId", target = "sourceLabel")* @Mapping(source = **"storageType"**, target = **"sourceId"**)  
 @Mapping(source = **"earchivedocType"**, target = **"docType"**)  
 *//@Mapping(source = "sourceId", target = "sourceId")* DrmDocument toDrmDocument(ESDocument esDocument);  
  
 */\*\*  
 \* Appends prefix to the document id  
 \*  
 \** ***@param value*** *id with prefix drm:  
 \** ***@return*** *a string  
 \*/* @Named(**"constructId"**)  
 **default** String constructId(String value) {  
  
 **if**(!(value.contains(**"earchive"**))) {  
 **return "drm\_"**.concat(value);  
 }  
 **return** value;  
 }  
  
 */\*\*  
 \* Finds corresponding enum value for the String  
 \*  
 \** ***@param value*** *status from the archive  
 \** ***@return*** *enum  
 \*/* @Named(**"toEnum"**)  
 **default** DocumentStatus toEnum(String value) {  
 **return** DocumentStatus.*getValues*(value);  
 }  
  
 */\*\*  
 \* Converts a string to uppercase  
 \*  
 \** ***@param value*** *to be converted  
 \** ***@return*** *a string  
 \*/* @Named(**"toUppercase"**)  
 **default** SecurityLevel toUppercase(String value) {  
 **return** Enum.*valueOf*(SecurityLevel.**class**, value.toUpperCase(Locale.***ENGLISH***));  
 }  
  
 */\*\*  
 \* Converts a string value to calendar type  
 \*  
 \** ***@param value*** *to be converted  
 \** ***@return*** *date time as calendar  
 \*/* @Named(**"toCalendar"**)  
 **default** Calendar toCalendar(String value) {  
 **try** {  
 Date date = **new** SimpleDateFormat(**"yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"**, Locale.***ENGLISH***).parse(value);  
 Calendar cal = **new** GregorianCalendar();  
 cal.setTime(date);  
 **return** cal;  
 } **catch** (ParseException e) {  
 **return null**;  
 }  
 }  
  
}