```java
@Slf4j
@Configuration
@EnableConfigurationProperties({MQConnectionProperties.class,
FTPSConnectionProperties.class})
@ComponentScan(basePackages = {"com.csg.springboot.platform.ssl"})
@Profile("D")
public class MQConfig_D {



@Bean
@ConditionalOnProperty(prefix = "platform.messaging.mq", name =
"autoconfigure", havingValue = "true")
public MQConnectionFactory connectionFactory(MQConnectionService
mqConnectionService, MQConnectionProperties mqConnectionProperties) throws
NamingException {
    return mqConnectionService.getConnectionFactory(mqConnectionProperties);
}
```

```java
@Data
@ConfigurationProperties(prefix = "platform.messaging.mq")
@NoArgsConstructor
@Profile("D")
public class MQConnectionProperties {

    @Value("${platform.messaging.mq.context-factory}")
    private String contextFactory;
    @Value("${platform.messaging.mq.provider-url}")
    private String providerUrl;
    @Value("${platform.messaging.mq.security-principal}")
    private String securityPrincipal;
    @Value("${platform.messaging.mq.security-credentials}")
    private String securityCredentials;
    @Value("${platform.messaging.mq.connection-factory}")
    private String connectionFactory;
    @Value("${platform.messaging.mq.keystore-password}")
    private String keystorePassword;
    @Value("${platform.messaging.mq.truststore-password}")
    private String truststorePassword;
    @Value("${platform.messaging.mq.keystore-path}")
    private String keystorePath;
    @Value("${platform.messaging.mq.truststore-path}")
    private String truststorePath;
    @Value("${platform.messaging.mq.keystore-type:pkcs12}")
    private String keystoreType;
    @Value("${platform.messaging.mq.truststore-type:jks}")
    private String truststoreType;
    @Value("${platform.messaging.mq.use-ibm-cipher-mappings:false}")
    private boolean useIbmCipherMappings;
    @Value("${platform.messaging.mq.autoconfigure}")
    private boolean autoconfigure;

    /**
     * Constructs an MQConnectionProperties object
     *
     * @param contextFactory       contextFactory
     * @param providerUrl          providerUrl
     * @param securityPrincipal    securityPrincipal
     * @param securityCredentials  securityCredentials
     * @param connectionFactory    connectionFactory
     * @param keystorePath         keystorePath
     * @param truststorePath       truststorePath
     * @param keystorePassword     keystorePassword
     * @param truststorePassword   truststorePassword
     * @param keystoreType         keystoreType
     * @param truststoreType       truststoreType
     * @param useIbmCipherMappings autoconfigure
     * @param useIbmCipherMappings useIBMCipherMappings
     */
public MQConnectionProperties(String contextFactory, String providerUrl,
String securityPrincipal, String securityCredentials, String
connectionFactory, String keystorePath, String truststorePath, String
keystorePassword, String truststorePassword, String keystoreType, String
truststoreType, boolean autoconfigure, boolean useIbmCipherMappings) {
        this.contextFactory = contextFactory;
        this.providerUrl = providerUrl;
        this.securityPrincipal = securityPrincipal;
        this.securityCredentials = securityCredentials;
        this.connectionFactory = connectionFactory;
        this.keystorePassword = keystorePassword;
        this.truststorePassword = truststorePassword;
        this.keystorePath = keystorePath;
        this.truststorePath = truststorePath;
        this.keystoreType = keystoreType;
        this.truststoreType = truststoreType;
        this.useIbmCipherMappings = useIbmCipherMappings;
        this.autoconfigure = autoconfigure;
    }
```

```
@Service
@Profile("D")
@Slf4j
public class MQConnectionService {

    private final SSLConfigurator sslConfigurator;

    /**
     * Creates a MQConfigurationService instance
     *
     * @param sslConfigurator The SSLConfigurator instance to use
     */
    public MQConnectionService(SSLConfigurator sslConfigurator) {
        this.sslConfigurator = sslConfigurator;
    }

    /**
     * Looks up and returns a MQConnectionFactory instance
     *
     * @param connectionProperties The MQ Connection Factory lookup
properties
     * @return The MQConnectionFactory instance looked up
     * @throws NamingException if a naming exception is encountered
     */
    public MQConnectionFactory getConnectionFactory(MQConnectionProperties
connectionProperties) throws NamingException {
        log.info("Inside the method  : getConnectionFactory");
        // Required to connect to MQ9+ server using either v7 or v9
libraries.
        if (connectionProperties.isUseIbmCipherMappings()) {
            System.setProperty("com.ibm.mq.cfg.useIBMCipherMappings",
"true");
        } else {
            System.setProperty("com.ibm.mq.cfg.useIBMCipherMappings",
"false");
        }

 Properties jndiProperties = setupInitialContextParams(connectionProperties);
 Context ctx = new InitialContext(jndiProperties);
MQConnectionFactory queueConnectionFactorySecure = (MQConnectionFactory)
ctx.lookup(connectionProperties.getConnectionFactory());
        SSLSocketFactory sslSocketFactory;
        sslSocketFactory =
sslConfigurator.createSSLContext(connectionProperties.getKeystorePath(),
connectionProperties.getTruststorePath(),
connectionProperties.getKeystorePassword(),
connectionProperties.getTruststorePassword(),
connectionProperties.getKeystoreType(),
connectionProperties.getTruststoreType()).getSocketFactory();
        queueConnectionFactorySecure.setSSLSocketFactory(sslSocketFactory);
        return queueConnectionFactorySecure;
    }

    /**
     * Constructs a map used to construct the InitialContext
     *
     * @param connectionProperties MQ Connection Factory lookup properties
     * @return Map with parameters related to the MQ CF lookup
     */
    private Properties setupInitialContextParams(MQConnectionProperties
connectionProperties) {
        log.info("Inside the method  : setupInitialContextParams");
        final Properties jndiProperties = new Properties();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
connectionProperties.getContextFactory());
        jndiProperties.put(Context.PROVIDER_URL,
connectionProperties.getProviderUrl());
        jndiProperties.put(Context.SECURITY_PRINCIPAL,
connectionProperties.getSecurityPrincipal());
        jndiProperties.put(Context.SECURITY_CREDENTIALS,
connectionProperties.getSecurityCredentials());
        return jndiProperties;
    }
```

Spring class

```
@Service
public class SSLConfigurator {
```

```
public SSLContext createSSLContext(String keystorePath, String
truststorePath, String keystorePassword, String trustStorePassword, String
keystoreType, String trustStoreType) {
    String message;
    try {
        log.debug("Creating SSLContext[v{},{}] with Keystore
[Path={},Type={}] and Truststore [Path={},Type={}]", new
Object[]{this.sslProtocolVersion, this.sslAlgorithm, keystorePath,
keystoreType, truststorePath, trustStoreType});
        SSLContext sslContext =
SSLContext.getInstance(this.sslProtocolVersion);
        KeyManagerFactory kmf =
KeyManagerFactory.getInstance(this.sslAlgorithm);
        TrustManagerFactory tmf =
TrustManagerFactory.getInstance(this.sslAlgorithm);
        KeyStore ks = KeyStore.getInstance(keystoreType);
        KeyStore ksTrust = KeyStore.getInstance(trustStoreType);
        InputStream is = this.getResourceInputStream(keystorePath);
        ks.load(is, keystorePassword.toCharArray());
        kmf.init(ks, keystorePassword.toCharArray());
        is = this.getResourceInputStream(truststorePath);
        ksTrust.load(is, trustStorePassword.toCharArray());
        tmf.init(ksTrust);
        sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), new
SecureRandom());
        return sslContext;
    } catch (KeyManagementException | KeyStoreException |
NoSuchAlgorithmException | UnrecoverableKeyException | CertificateException |
IOException var13) {
        message = String.format("Unexpected exception [%s] while creating SSL
Context", var13.getClass().getName());
        log.error(message, var13);
        throw new SSLContextCreationException(message, var13);
    } catch (MissingResourceException var14) {
        message = String.format("Resource reference [%s] cannot be
resolved.", var14.getKey());
        log.error(message, var14);
        throw new SSLContextCreationException(message, var14);
    }
}
```

```
@EnableJms
@SpringBootApplication
@EnableTransactionManagement
@Slf4j
@Data
public class Bloker {
    private SslParams params;

    public Broker(SslParams params) {
        this.params = params;
    }

    public static void main(String[] args) {
        SpringApplication.run(Bloker.class, args);
    }

    @PostConstruct
    void postConstruct() {
        setTrustStoreParams();
    }

    private void setTrustStoreParams() {
        log.info(String.format("Setting javax properties"));
        System.setProperty("javax.net.ssl.trustStore",
params.trustStorePath);
        System.setProperty("javax.net.ssl.trustStorePassword",
params.trustStorePassword);
    }
}
```

```
platform:
  messaging:
    mq:
      context-factory: com.sun.jndi.ldap.LdapCtxFactory
      truststore-path: classpath:pki/ca_cs_combined.jks
      truststore-password: ca_cs_combined
      autoconfigure: true
      truststoreType: jks
      useIbmCipherMappings: false
      provider-url: ldap://omb-gldap-..:40389/ou=cf,ou=ch-dz-
z,ou=omb,ou=applications,dc=intranet,dc=net
      security-principal:
      security-credentials:
      connection-factory: cn=CH1A01
      keystore-path: classpath:pki/S20XXXX_T.p12
      keystore-password: ANY
      keystoreType: pkcs12
```