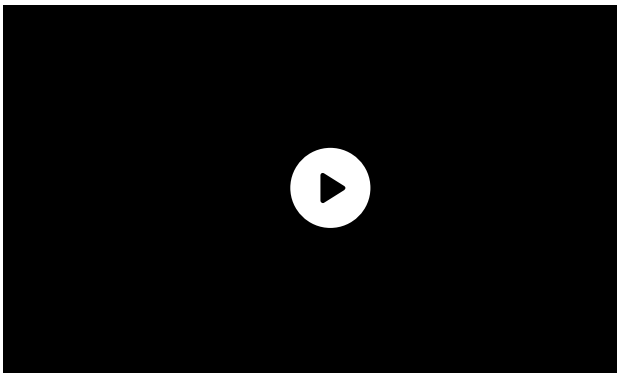
(/)

RSA in Java



Last updated: June 12, 2024

Written by: Krzysztof Majewski (https://www.baeldung.com/author/krzysztofmajewski)

Reviewed by: Michal Aibin (https://www.baeldung.com/editor/michal-author)

Security (https://www.baeldung.com/category/security)

1. Introduction

RSA, or in other words Rivest–Shamir–Adleman (/cs/rsa-public-key-format), is an asymmetric cryptographic algorithm. It differs from symmetric algorithms like DES (https://en.wikipedia.org/wiki/Data_Encryption_Standard) or AES (/java-aes-encryption-decryption) by having two keys. A public key that we can share with anyone is used to encrypt data. And a private one that we keep only for ourselves and it's used for decrypting the data

In this tutorial, we'll learn how to generate, store and use the RSA keys in Java.

2. Generate RSA Key Pair

Before we start the actual encryption, we need to generate our RSA key pair. We can easily do it by using the *KeyPairGenerator* from *java.security* package:

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
generator.initialize(2048);
KeyPair pair = generator.generateKeyPair();
```

The generated key will have a size of 2048 bits.

Next, we can extract the private and public key:

```
PrivateKey privateKey = pair.getPrivate();
PublicKey publicKey = pair.getPublic();
```

We'll use the public key to encrypt the data and the private one for decrypting it.

3. Storing Keys in Files

Storing the key pair in memory is not always a good option. Mostly, the keys will stay unchanged for a long time. In such cases, it's more convenient to store them in files.

To save a key in a file, we can use the *getEncoded* method, which returns the key content in its primary encoding format:

```
try (FileOutputStream fos = new FileOutputStream("public.key")) {
    fos.write(publicKey.getEncoded());
}
```

To read the key from a file, we'll first need to load the content as a byte array:

```
File publicKeyFile = new File("public.key");
byte[] publicKeyBytes = Files.readAllBytes(publicKeyFile.toPath());
```

and then use the *KeyFactory* to recreate the actual instance:

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(publicKeyBytes);
keyFactory.generatePublic(publicKeySpec);
```

The key byte content needs to be wrapped with an *EncodedKeySpec* class. Here, we're using the *X509EncodedKeySpec*, which represents the default algorithm for *Key::getEncoded* method we used for saving the file.

In this example, we saved and read only the public key file. The same steps can be used for handling the private key.

Remember, keep the file with a private key as safe as possible with access as limited as possible. Unauthorized access might bring security issues.

4. Working With Strings

Now, let's take a look at how we can encrypt and decrypt simple strings. Firstly, we'll need some data to work with:

```
String secretMessage = "Baeldung secret message";
```

Secondly, we'll need a *Cipher* (/java-cipher-class) object initialized for encryption with the public key that we generated previously:

```
Cipher encryptCipher = Cipher.getInstance("RSA");
encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);
```

Having that ready, we can invoke the *doFinal* method to encrypt our message. Note that it accepts only byte array arguments, so we need to transform our string before:

```
byte[] secretMessageBytes = (/)
secretMessage.getBytes(StandardCharsets.UTF_8);)
byte[] encryptedMessageBytes =
encryptCipher.doFinal(secretMessageBytes);
```

Now, our message is successfully encoded. If we'd like to store it in a database or send it via REST API (/rest-with-spring-series), it would be more convenient to encode it with the Base64 Alphabet (/java-base64-encode-and-decode):

```
String encodedMessage =
Base64.getEncoder().encodeToString(encryptedMessageBytes);
```

This way, the message will be more readable and easier to work with.

Now, let's see how we can decrypt the message to its original form. For this, we'll need another *Cipher* instance. This time we'll initialize it with a decryption mode and a private key:

```
Cipher decryptCipher = Cipher.getInstance("RSA");
decryptCipher.init(Cipher.DECRYPT_MODE, privateKey);
```

We'll invoke the cipher as previously with the *doFinal* method:

```
byte[] decryptedMessageBytes =
decryptCipher.doFinal(encryptedMessageBytes);
String decryptedMessage = new String(decryptedMessageBytes,
StandardCharsets.UTF_8);
```

Finally, let's verify if the encryption-decryption process went correctly:

```
assertEquals(secretMessage, decryptedMessage);
```

5. Working With Files

It is also possible to encrypt whole files. As an example, let's create a temp file with some text content:

```
Path tempFile = Files.createTempFile("temp", "txt");
Files.writeString(tempFile, "some secret message");
```

Before we start the encryption, we need to transform its content into a byte array:

```
byte[] fileBytes = Files.readAllBytes(tempFile);
```

Now, we can use the encryption cipher:

```
Cipher encryptCipher = Cipher.getInstance("RSA");
encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);
byte[] encryptedFileBytes = encryptCipher.doFinal(fileBytes);
```

And finally, we can overwrite it with new, encrypted content:

```
try (FileOutputStream stream = new FileOutputStream(tempFile.toFile()))
{
    stream.write(encryptedFileBytes);
}
```

The decryption process looks very similar. The only difference is a cipher initialized in decryption mode with a private key:

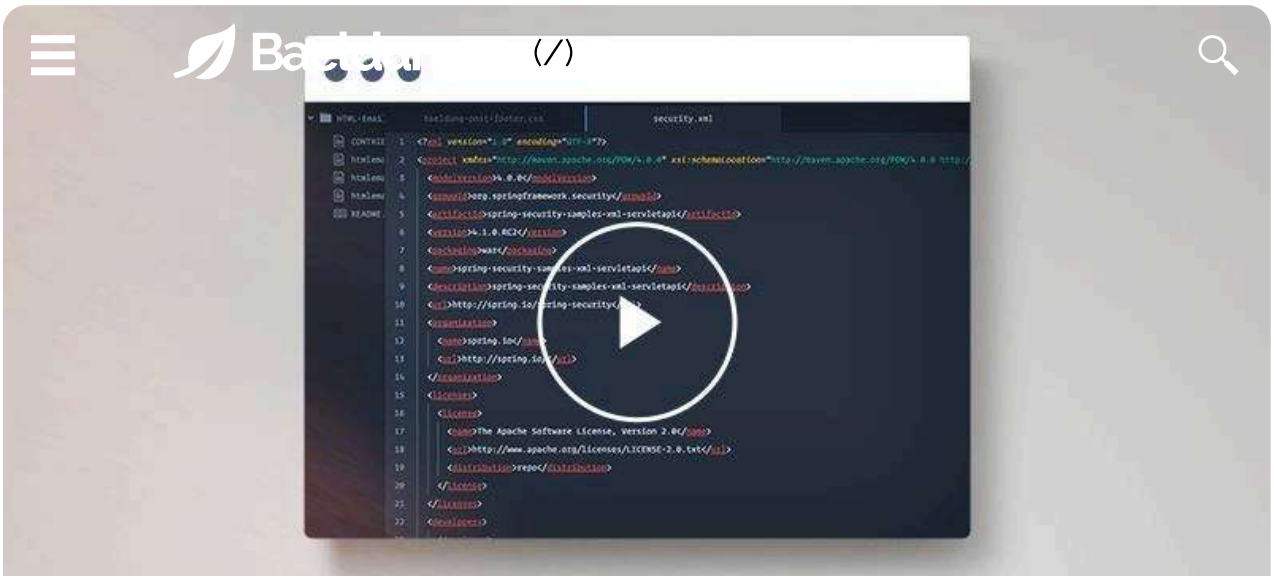
```
byte[] encryptedFileBytes = Files.readAllBytes(tempFile);
Cipher decryptCipher = Cipher.getInstance("RSA");
decryptCipher.init(Cipher.DECRYPT_MODE, privateKey);
byte[] decryptedFileBytes = decryptCipher.doFinal(encryptedFileBytes);
try (FileOutputStream stream = new FileOutputStream(tempFile.toFile()))
{
    stream.write(decryptedFileBytes);
}
```

As the last step, we can verify if the file content matches the original value:

```
String fileContent = Files.readString(tempFile);
Assertions.assertEquals("some secret message", fileContent);
```

6. Summary

In this article, we've learned how to create RSA keys in Java and how to use them to encrypt and decrypt messages and files. As always, all source code is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-security-algorithms).



Learn the basics of securing a REST API with Spring

Get access to the video lesson (/security-video-guide)

2 COMMENTS

Oldest

View Comments

(/)

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
ALL BULK COURSES (/COURSES/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/COURSES/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS/)