


Computer science → Backend → Spring Boot → Spring Security

Spring Security Crypto

Theory

Practice

 10% completed, 0 problems solved

▼

Theory

🕒 21 minutes reading

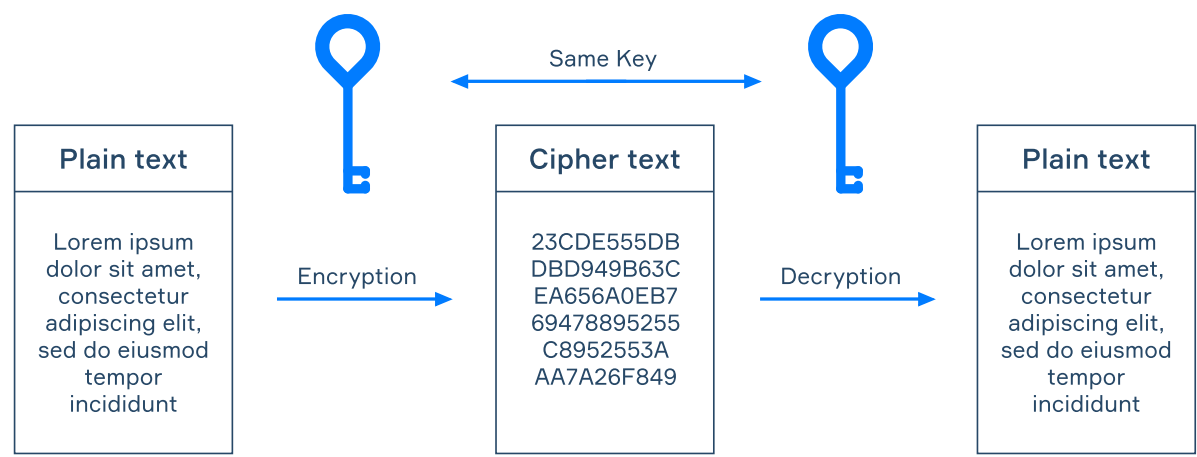
Verify to skip

Start practicing

When developing a backend application, you must know how to store user information in the database in a secure manner. It doesn't matter whether you are a freelancer or a multi-billion dollar corporation – you need to assume that somebody may one day break into your database. You are always exposed to the risks of a hacker attack from the outside or a data leak from the inside.

§1. Encryption and hashing principles

One of the ways to protect personal data is **encryption**. There are two kinds of encryption: **symmetric** and **asymmetric**. The major difference between them is in the keys used to encrypt and decrypt data. In this topic, we'll focus only on the symmetric encryption algorithm. Unlike asymmetric encryption, it uses the same key to encrypt and decrypt data. One of the most widely known and simple examples of symmetric encryption algorithms is [Caesar Cipher](#). You should always remember that everyone with a key can decrypt information. So, for data safety, it's important to properly store the keys in a secure place and prevent unwanted access. With personal data, be mindful of mandatory requirements for its storage and encryption that may be foreseen by legislation. For instance, storing credit card information falls under the [PCI DSS](#) rules, a regulation established by the payment systems applied internationally.




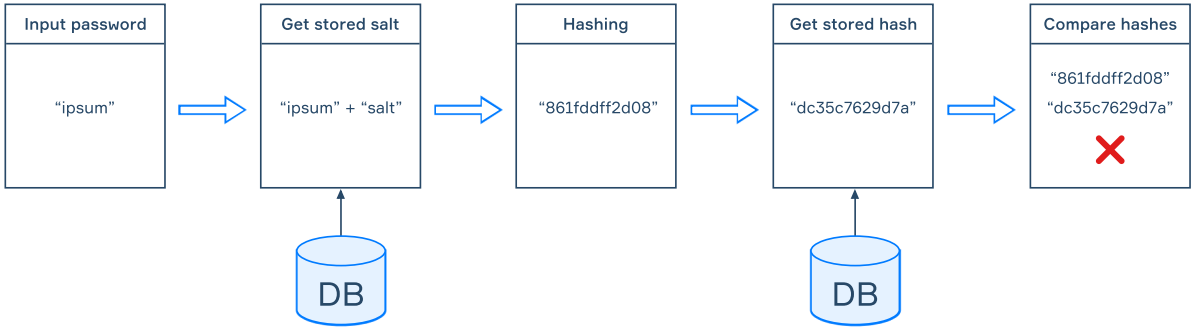
As for the passwords, in general, they should be **hashed**. Every time a user logs into the system, we will hash their password and compare it with the stored hash in the database. For this reason, in cases where the user forgets the password, they can only reset it, but not retrieve the original password from the system. This ensures that even if the attackers get access to the password hashes list, they can't use them directly. Nevertheless, there is still the risk of the hashed passwords getting hacked through a brute force or [rainbow table attack](#). To slow down the brute force, you can use a solid hashing algorithm, which makes it too time-consuming for hackers. As for the rainbow table, adding "**salt**" to the password before hashing protects against it. By salt here we mean a string that is usually stored in the database along with the hashed password. The idea of the salt is to generate different hashes for the same passwords. For better protection, it is best to use a random and long enough string as the salt.

2 required topics

-  [Spring components](#) ▼
-  [Insecure Cryptographic Storage](#) ▼

1 dependent topic

-  [Authentication](#) ▼



Hashing, salting, and encrypting are very complex procedures. Some bugs in the implementation can lead to critical consequences, so it's better to leave it to security experts. In Spring Security, you can find the **Spring Crypto** module that includes encryptors, key generators, and password encoders, making it easier to use in your system. In the following sections, we'll describe each of the elements of this module in more detail.

§2. Adding Spring Crypto to your project

If you are using Spring Boot Security starter, you don't need to add any additional dependencies to use Spring Crypto: they are already included.

▼ Gradle Java

```
1 dependencies {
2     implementation 'org.springframework.boot:spring-boot-starter-
security:2.7.2'
3 }
```

▼ Gradle Kotlin

```
1 dependencies {
2     implementation("org.springframework.boot:spring-boot-starter-
security:2.7.2")
3 }
```

▼ Maven

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-security</artifactId>
4     <version>2.7.2</version>
5 </dependency>
6
```

You can also add Spring Crypto as a separate dependency, even without using the whole Spring Framework:

▼ Gradle Java

```
1 dependencies {
2     implementation 'org.springframework.security:spring-security-
crypto:5.7.2'
3 }
```

▼ Gradle Kotlin

```
1 dependencies {
2     implementation("org.springframework.security:spring-security-
crypto:5.7.2")
3 }
```

▼ Maven

```

1  <dependency>
2    <groupId>org.springframework.security</groupId>
3    <artifactId>spring-security-crypto</artifactId>
4    <version>5.7.2</version>
5  </dependency>

```

§3. Encryptors and key generators

Spring Crypto has two main interfaces for encryptors:

- `BytesEncryptor`
- `TextEncryptor`

The first one declares the encryption and decryption of the byte array. The second one will work with `String`. Any implementation of both interfaces can be done with a static method of the `Encryptors` class.

Declare `AesBytesEncryptor` as a bean:

▼ Java

```

1  @SpringBootApplication
2  public class Application {
3
4      public static void main(String[] args) {
5          SpringApplication.run(Application.class);
6      }
7
8      @Bean
9      public BytesEncryptor aesBytesEncryptor() {
10
11          String password = "hackme"; // should be kept in a secure
place and not be shared
12
13          String salt = "8560b4f4b3"; // should be hex-encoded with
even number of chars
14
15          return Encryptors.standard(password, salt);
16      }
17  }

```

▼ Kotlin

```

1  @Bean
2  open fun aesBytesEncryptor(): BytesEncryptor {
3      val password = "hackme" // should be kept in a secure place and
not be shared
4      val salt = "8560b4f4b3" // should be hex-encoded with even
number of chars
5      return Encryptors.standard(password, salt)
6  }

```

Below we are using the default `BytesEncryptor`, which uses 256-bit AES symmetric encryption algorithm. This is one of the most secure algorithms that has been approved by the U.S. government and applied to store top-secret information.

Let's see how to use the encryptor:

▼ Java

```

1  @SpringBootApplication
2  public class Application implements CommandLineRunner {
3

```

```

4     public static void main(String[] args) {
5         SpringApplication.run(Application.class);
6     }
7
8     @Bean
9     public BytesEncryptor aesBytesEncryptor() { ... }
10
11
12     @Override
13
14     public void run(String... args) throws Exception {
15
16         BytesEncryptor bytesEncryptor = aesBytesEncryptor();
17
18
19         byte[] inputData = {104, 121, 112, 101, 114, 115, 107, 105,
108, 108};
19
20         byte[] encryptedData = bytesEncryptor.encrypt(inputData);
21
22         byte[] decryptedData =
23 bytesEncryptor.decrypt(encryptedData);
24
25
26         System.out.println("Input data: " + new String(inputData));
27
28         System.out.println("Encrypted data: " + new
29 String(encryptedData));
30
31         System.out.println("Decrypted data: " + new
32 String(decryptedData));
33     }
34 }

```

▼ Kotlin

```

1  val inputData = arrayOf(104, 121, 112, 101, 114, 115, 107, 105,
108, 108)
2  val encryptedData = bytesEncryptor.encrypt(inputData)
3  val decryptedData = bytesEncryptor.decrypt(encryptedData)
4
5  println("Input data: $inputData")
6  println("Encrypted data: $encryptedData")
7  println("Decrypted data: $decryptedData")

```

The output will be:

```

Input data: hyperskill
Encrypted data: *jS*) ^?#| d-%
Decrypted data: hyperskill

```

Most of the time, data that needs to be encrypted can be represented as `String`. In that case, it's better to use `TextEncryptor`. It has two implementations:

- `NoOpTextEncryptor`
- `HexEncodingTextEncryptor`

`NoOp` stands for "No Operation", so this encryptor does nothing and doesn't really interest us.

`HexEncodingTextEncryptor` uses the previously discussed `BytesEncryptor` under the hood. The encrypted data will be hex-encoded, so it's easier to store it in the file system or database.

Define `HexEncodingTextEncryptor`:

▼ Java

```
1  @Bean
2  public TextEncryptor hexEncodingTextEncryptor() {
3      String password = "hackme"; // should be kept in a secure place
and not be shared
4      String salt = "8560b4f4b3"; // should be hex-encoded with even
number of chars
5      return Encryptors.text(password, salt);
6  }
```

▼ Kotlin

```
1  @Bean
2  open fun hexEncodingTextEncryptor(): TextEncryptor {
3      val password = "hackme" // should be kept in a secure place and
not be shared
4      val salt = "8560b4f4b3" // should be hex-encoded with even
number of chars
5      return Encryptors.text(password, salt)
6  }
```

Use it to encode the text:

▼ Java

```
1  String inputData = "hyperskill";
2  String encryptedData = textEncryptor.encrypt(inputData);
3  String decryptedData = textEncryptor.decrypt(encryptedData);
4
5  System.out.println("Input data: " + inputData);
6  System.out.println("Encrypted data: " + encryptedData);
7  System.out.println("Decrypted data: " + decryptedData);
```

▼ Kotlin

```
1  val inputData = "hyperskill"
2  val encryptedData = textEncryptor.encrypt(inputData)
3  val decryptedData = textEncryptor.decrypt(encryptedData)
4
5  println("Input data: $inputData")
6  println("Encrypted data: $encryptedData")
7  println("Decrypted data: $decryptedData")
```

Here is the output:

```
Input data: hyperskill
Encrypted data:
ec2f77cff5a92723a9a101193f33937068802fb0dda2d70df440dc35c7629d7a
Decrypted data: hyperskill
```

As you can see in the examples above, we hard-coded our "salt" value in the encryptor bean. However, you can also use mechanisms provided by Spring, which is even better because then you'll use a randomly generated "salt":

▼ Java

```
1  String salt = KeyGenerators.string().generateKey();
```

▼ Kotlin

```
1 | val salt = KeyGenerators.string().generateKey()
```

This line will generate a hex-encoded key that is 8 bytes in length:

```
861fddff2d08c730
```

§4. Passwords encoders

Passwords are hashed by the use of `PasswordEncoder`. This interface declares methods to encode input and match hash with raw password. Let's see how to use the most simple encoder implementation `NoOpPasswordEncoder`, which, as you can understand from its name, leaves the password as it is.

▼ Java

```
1 | PasswordEncoder noOpEncoder = NoOpPasswordEncoder.getInstance();
2 |
3 | String rawPassword = "hackme";
4 | String encodedPassword = noOpEncoder.encode("hackme");
5 | boolean isMatched = noOpEncoder.matches(rawPassword,
encodedPassword);
6 |
7 | System.out.println("Encoded password: " + encodedPassword);
8 | System.out.println("Match: " + isMatched);
```

▼ Kotlin

```
1 | val noOpEncoder = NoOpPasswordEncoder.getInstance()
2 |
3 | val rawPassword = "hackme"
4 | val encodedPassword = noOpEncoder.encode("hackme")
5 | val isMatched = noOpEncoder.matches(rawPassword, encodedPassword)
6 |
7 | println("Encoded password: $encodedPassword")
8 | println("Match: $isMatched")
```

Here is the output:

```
Encoded password: hackme
Match: true
```

This encoder is not secure, which means it is deprecated. It is provided in the library for legacy and testing purposes only. These are the recommended implementations:

- `BCryptPasswordEncoder`
- `PbkdfPasswordEncoder`
- `SCryptPasswordEncoder`

Take a closer look at `BCryptPasswordEncoder`:

▼ Java

```
1 | int strength = 7;
2 | PasswordEncoder bCryptEncoder = new BCryptPasswordEncoder(strength);
3 |
4 | String rawPassword = "hackme";
5 | String firstEncodedPassword = bCryptEncoder.encode(rawPassword);
6 | String secondEncodedPassword = bCryptEncoder.encode(rawPassword);
7 |
8 | System.out.println("First encoded password: " +
firstEncodedPassword);
```

```
9 System.out.println("Second encoded password: " +
secondEncodedPassword);
```

▼ Kotlin

```
1 val strength = 7
2 val bCryptEncoder = BCryptPasswordEncoder(strength)
3
4 val rawPassword = "hackme"
5 val firstEncodedPassword = bCryptEncoder.encode(rawPassword)
6 val secondEncodedPassword = bCryptEncoder.encode(rawPassword)
7
8 println("First encoded password: $firstEncodedPassword")
9 println("Second encoded password: $secondEncodedPassword")
```

As you can see, we are passing the parameter to the encoder constructor. This is one of the "strengths" of the `bCrypt` algorithm: log rounds to use. In this way, the algorithm allows the system able to adapt to more powerful computers, so more work has to be done to calculate the hash. Try to use a higher number and see how it will affect the performance.

And the output will be the following:

```
First encoded password:
$2a$07$Q/XxLrLPHniRG19PNSLiSuPGcDkbv54U3.DAEQXRpSd5qKXpCk2V2
Second encoded password:
$2a$07$.8cNtKbeAqDrxsPAquxRRuSYpg6RKCKqxoW700xxht7yKLPUcXQye
```

We passed the same password to both `encode` calls but got completely different results. This is because the `bCrypt` algorithm implementation in Spring will manage "salt" for you.

There is one more implementation left, namely `DelegatingPasswordEncoder`. It acts as a facade to all password encoding algorithms and makes it possible to select an algorithm at runtime. The algorithm that was used is stored with a password in the following format: `{id}encodedPassword`. Below you can find examples of passwords encoded with `DelegatingPasswordEncoder`.

Id	Implementation	Example
bcrypt	BCryptPasswordEncoder	{bcrypt}\$2a\$10\$dXJ3SW6G7P50IGmMkkmwe.20cQQubK3.HZWzG3YB1tlRy.fqvM/BG
noop	NoOpPasswordEncoder	{noop}password
pbkdf2	Pbkdf2PasswordEncoder	{pbkdf2}5d923b44a6da29f3ddf3e3c8d29412723dcbde72445e8f6bf3b508fbf17fa4ed4d6b99ca763d8dc
scrypt	SCryptPasswordEncoder	{scrypt}\$e0801\$8bWJaSu2IKSn9Z9kM+TPXfOc/9bdYSrN10D9qfVThWEwdRTnO7re7Ei+fUZRJ68k9ITyuTeUp4of424hHnazw== \$OAOec05+bXxvuu/1qZ6NUR+xQYvYv7BeL1QxwRpY5Pc=
sha256	StandardPasswordEncoder	{sha256}97cde38028ad898ebc02e690819fa220e88c62e0699403e94fff291cfffaf8410849f27605abcbcb0

§5. Conclusion

Now you know about encryptors, key generators, and passwords that compose the Spring Crypto module of the Spring Security library and how to use them.

Remember, if you store users' passwords somewhere, they should be hashed. Which data should be encrypted is more difficult to decide as there is no straight answer. A good idea is to perform a risk assessment about possible attack threats and check the regulatory requirements.

A variety of complex algorithms has already been implemented for you by Spring developers. You can choose the one that suits your needs most and provides better security to keep valuable information safe.

Table of contents:

- [1 Spring Security Crypto](#)
- [§1. Encryption and hashing principles](#)
- [§2. Adding Spring Crypto to your project](#)
- [§3. Encryptors and key generators](#)
- [§4. Passwords encoders](#)
- [§5. Conclusion](#)
- [Discussion](#)

Report a typo

41 users liked this piece of theory. 0 didn't like it. What about you?



Start practicing


Verify to skip

This content was created 6 months ago and updated 4 days ago. [Share your feedback below in comments to help us improve it!](#)

[Comments \(11\)](#)

[Useful links \(1\)](#)

[Show discussion](#)

JetBrains Academy

Tracks


Pricing

For organizations

About

Contribute

Careers

Become beta tester

Be the first to see what's new