

Spring Security

1. Implementar Spring Security versión 6 para spring boot 3.

Encoders más utilizados en Spring Security:

Algoritmo	Seguridad (2025)	Resistencia GPU/ASIC	Rendimiento	Uso de memoria	Sal y parámetros	Prefijo del hash	Soporte Spring Boot 3+	¿Cuándo usar?
BCrypt	Alta	Alta	Muy bueno	Bajo	Sal automática; cost (work factor)	\$2a\$, \$2b\$ o {bcrypt}	Excelente (por defecto recomendado)	Proyectos generales; balance seguridad/rendimiento
Argon2id	Muy alta	Muy alta	Medio	Alto	Memoria, iteraciones y paralelismo	\$argon2id\$ o {argon2}	Excelente	Datos sensibles; entornos modernos con buen hardware
PBKDF2 (SHA-256)	Media-Alta	Media	Alto (rápido)	Bajo	Iteraciones (work factor)	{pbkdf2}	Excelente	Compatibilidad corporativa/legacy; FIPS-friendly
SCrypt	Muy alta	Muy alta	Medio-bajo	Alto	N (CPU/mem), r, p	{scrypt}	Buena	Alternativa a Argon2 cuando se prefiere scrypt

¿Qué es BCrypt?

BCrypt es un algoritmo de **hash seguro** diseñado específicamente para proteger **contraseñas**. A diferencia de un cifrado tradicional, **no puede revertirse** (no se puede “desencriptar”), sino que sirve para **verificar coincidencias**.

Su principal fortaleza es que **consume tiempo y recursos controladamente**, lo que dificulta ataques por fuerza o con hardware especializado (GPU, ASIC, etc.).

¿Cómo funciona BCrypt paso a paso?

1. Generación de “salt” aleatoria

- Antes de generar el hash, BCrypt crea una **salt** (una cadena aleatoria de 16 bytes).
- Este salt se combina con la contraseña original.
- Gracias a esto, **dos usuarios con la misma contraseña tendrán hashes distintos**.

Ej:

“contraseña”: “cliente123”

“salt”:

“\$2a\$10\$ElpqI8bM1k7piw9COv5/ROSreFGaBeCoq5OmvCgWzt5G4H4/eaA.i”,

Algoritmo de derivación (basado en Blowfish)

- BCrypt deriva la contraseña y el salt con un algoritmo basado en **Blowfish**.
- Aplica miles de ciclos de cálculo llamadas **cost factor** (por defecto 10 → $2^{10} = 1024$ iteraciones).
- Este tiempo intencionado permite que **cada hash tarde ~100–300 ms**, lo cual es poco para el usuario, pero bastante caro para un atacante.

Resultado final

- Devuelve una cadena codificada en Base64 que incluye:
 - el tipo de algoritmo (\$2a\$, \$2b\$\$, \$2y\$),
 - el cost factor (por ejemplo \$10\$),
 - el salt y el hash mezclados.

\$2a\$10\$u4nYjZ7NPaXCP4oVxReKyeWv1uk8qxk.ZlFpuBExKOrM0Z86dax.i
↑ ↑ ↑ ↑ _____
| | | | └ hash + salt codificados
| | | | └ 10 → cost factor ($2^{10} = 1024$ rondas)
| | | | └ versión del algoritmo (2a, 2b, 2y)
| | | | └ indicador de BCrypt

¿Cómo se verifica una contraseña?

Cuando el usuario inicia sesión:

1. Spring obtiene el hash almacenado, por ejemplo:
2. \$2a\$10\$u4nYjZ7NPaXCP4oVxReKyeWv1uk8qxk.ZlFpuBExKOrM0Z86dax.i
3. Extrae de ahí el **salt** y el **cost factor**.
4. Aplica el mismo proceso de hash a la contraseña que envió el usuario.
5. Si el resultado **coincide con el hash guardado**, la contraseña es válida.

Este proceso se realiza con el método:

passwordEncoder.matches(contraseñaIngresada, hashGuardado)

2. Desde spring boot en el archivo pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-crypto</artifactId>
</dependency>
```

3. Crear carpeta webconfig y archivo SecurityConfig.java:

```

in > java > com > vivitasol > projectbackend > webconfig > J SecurityConfig.java > Language Support for Java(TM) by Red Hat
  1 package com.vivitasol.projectbackend.webconfig;
  2 import org.springframework.context.annotation.Bean;
  3 import org.springframework.context.annotation.Configuration;
  4 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
  5 import org.springframework.security.web.SecurityFilterChain;
  6
  7 @Configuration
  8 public class SecurityConfig {
  9
 10     @Bean
 11     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
 12         http.csrf(csrf -> csrf.disable())
 13             .authorizeHttpRequests(requests -> requests
 14                 .anyRequest().permitAll());
 15         return http.build();
 16     }
 17 }

```

4. Agregar método que permita encriptar contraseña:

```

@Configuration
public class SecurityConfig {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(requests -> requests
                .anyRequest().permitAll());
        return http.build();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}

```

Verificar este import:

```
import org.springframework.security.crypto.password.PasswordEncoder;
```

5. Revisamos desde postman:

Petición post: <http://localhost:8080/api/usuarios>

The screenshot shows the Postman interface with a successful POST request to `http://localhost:8080/api/usuarios`. The request body is a JSON object:

```

1 {
2     "username": "cliente2",
3     "password": "cliente23",
4     "email": "cliente2@gmail.com",
5     "rol": "CLIENTE",
6     "enabled": true
7 }

```

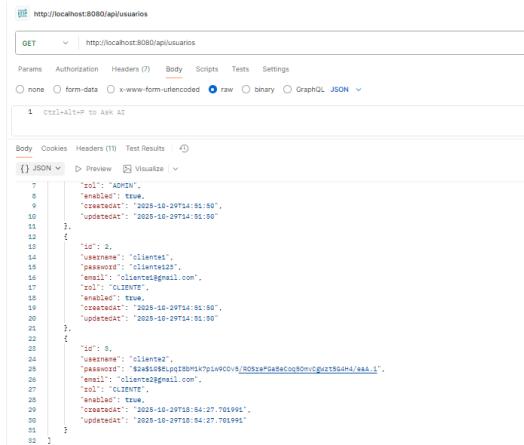
The response status is `200 OK`, and the response body is:

```

1 {
2     "id": 3,
3     "username": "cliente2",
4     "password": "$2a$10$EpQoIBm1k7piw9CoV6/RO$zrFGaBeCoq5OnVcgIzt5G4H4/eaA.i",
5     "email": "cliente2@gmail.com",
6     "rol": "CLIENTE",
7     "enabled": true,
8     "createdAt": "2025-10-29T18:54:27.701999Z",
9     "updatedAt": "2025-10-29T18:54:27.701999Z"
10 }

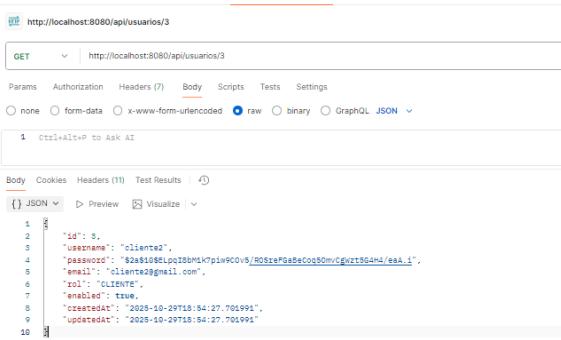
```

Petición get: http://localhost:8080/api/usuarios



```
http://localhost:8080/api/usuarios
GET http://localhost:8080/api/usuarios
Params Authorization Headers (7) Body Scripts Tests Settings
○ none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL JSON
1 Ctrl+Alt+P to Ask AI
Body Cookies Headers (11) Test Results ⓘ
{} JSON ▶ Preview ⚡ Visualize ▾
7
8   "id": "5e71a547-4a7c-4a2d-8f3a-14a2a2a2a2a2",
9   "enabled": true,
10  "createdAt": "2028-10-29T14:51:50",
11  "updatedAt": "2028-10-29T14:51:50"
12 [
13   {
14     "id": 2,
15     "username": "cliente1",
16     "password": "cliente123",
17     "email": "cliente1@gmail.com",
18     "rol": "CLIENTE",
19     "enabled": true,
20     "createdAt": "2028-10-29T14:51:50",
21     "updatedAt": "2028-10-29T14:51:50"
22   },
23   {
24     "id": 3,
25     "username": "cliente2",
26     "password": "$2a$10$8LpqZ5DMLK7piw9COv5/$0$re#G$e$C$0$0$mv/C$u$vt$6$e$H/e$aa.i",
27     "email": "cliente2@gmail.com",
28     "rol": "CLIENTE",
29     "enabled": true,
30     "createdAt": "2028-10-29T18:54:27.701991",
31     "updatedAt": "2028-10-29T18:54:27.701991"
32 }
]
```

Petición get: http://localhost:8080/api/usuarios/3



```
http://localhost:8080/api/usuarios/3
GET http://localhost:8080/api/usuarios/3
Params Authorization Headers (7) Body Scripts Tests Settings
○ none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary ○ GraphQL JSON
1 Ctrl+Alt+P to Ask AI
Body Cookies Headers (11) Test Results ⓘ
{} JSON ▶ Preview ⚡ Visualize ▾
1
2   "id": 3,
3   "username": "cliente2",
4   "password": "$2a$10$8LpqZ5DMLK7piw9COv5/$0$re#G$e$C$0$0$mv/C$u$vt$6$e$H/e$aa.i",
5   "email": "cliente2@gmail.com",
6   "rol": "CLIENTE",
7   "enabled": true,
8   "createdAt": "2028-10-29T18:54:27.701991",
9   "updatedAt": "2028-10-29T18:54:27.701991"
10 }
```