Introduction to the Intermediate Class

The Intermediate Class

The Intermediate Class contains the following 5 modules:

- Password Storage (5 lessons)
- Spring Security Advanced Configuration (4 lessons)
- Advanced Authentication (3 lessons)
- Advanced Authorization (4 lessons)
- Basic REST API Security (4 lessons)

Each Module is organized in several video Lessons - containing lesson notes and other resources.

The material is meant to be experienced as video, and **the lesson notes should be used as a reference not as comprehensive documentation**.

Finally, have a look at the Course Intro page for additional, course-wide details.

Resources

The git repository of the project, hosted over on Github:

- The Module 6 branch

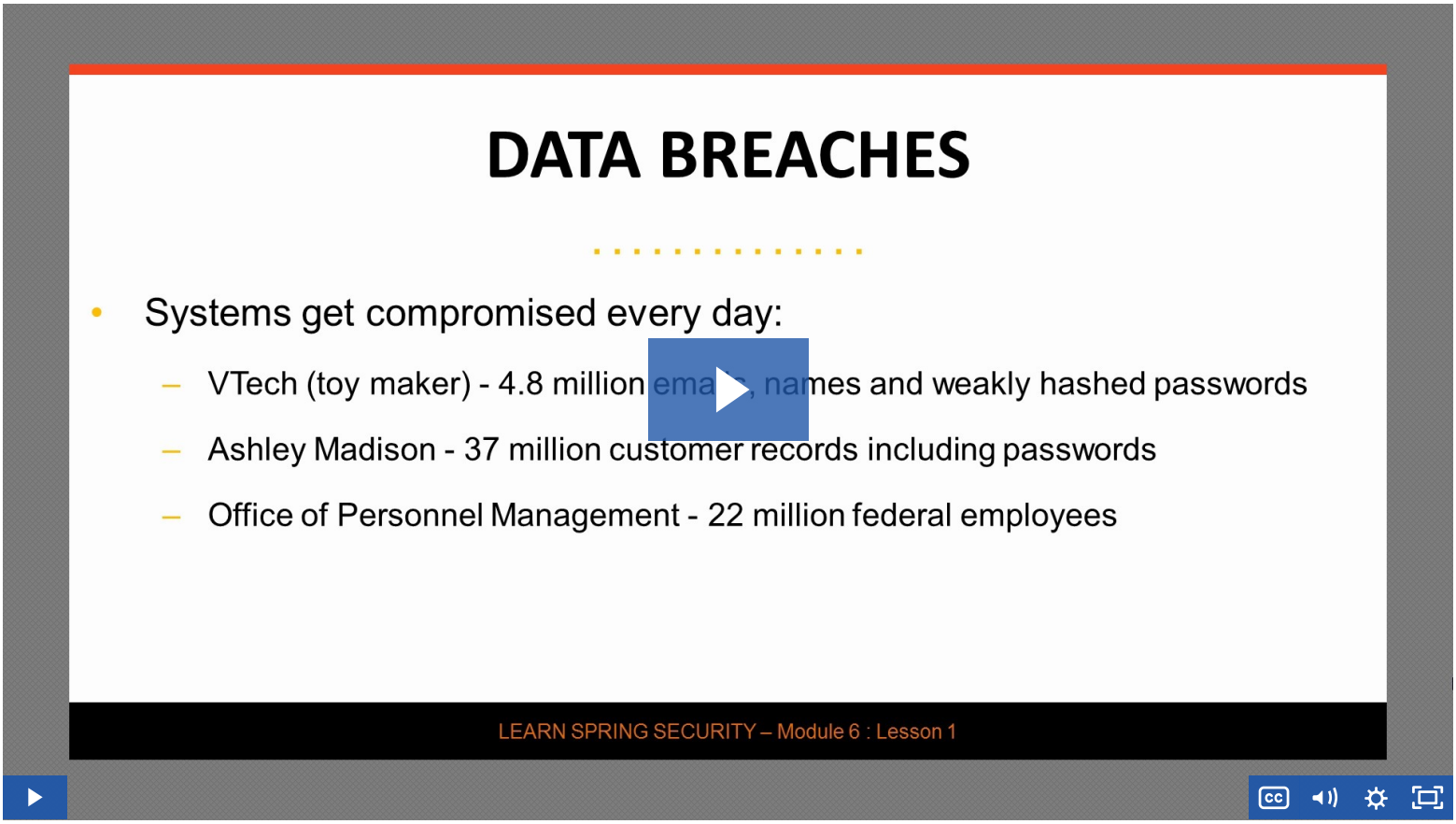
- The Module 7 branch

- The Module 8 branch

- The Module 9 branch

- Module 10 is theoretical

Lesson 1: Introduction to Storing Passwords



1. Goal

The goal of this lesson is to guide you through the foundations of how to properly and correctly store credentials.

2. Lesson Notes

If you're using the git repository to get the project - you should be using the module6 branch.

If you want to revisit how the Intermediate Class is structured, have a quick look at the previous lesson.

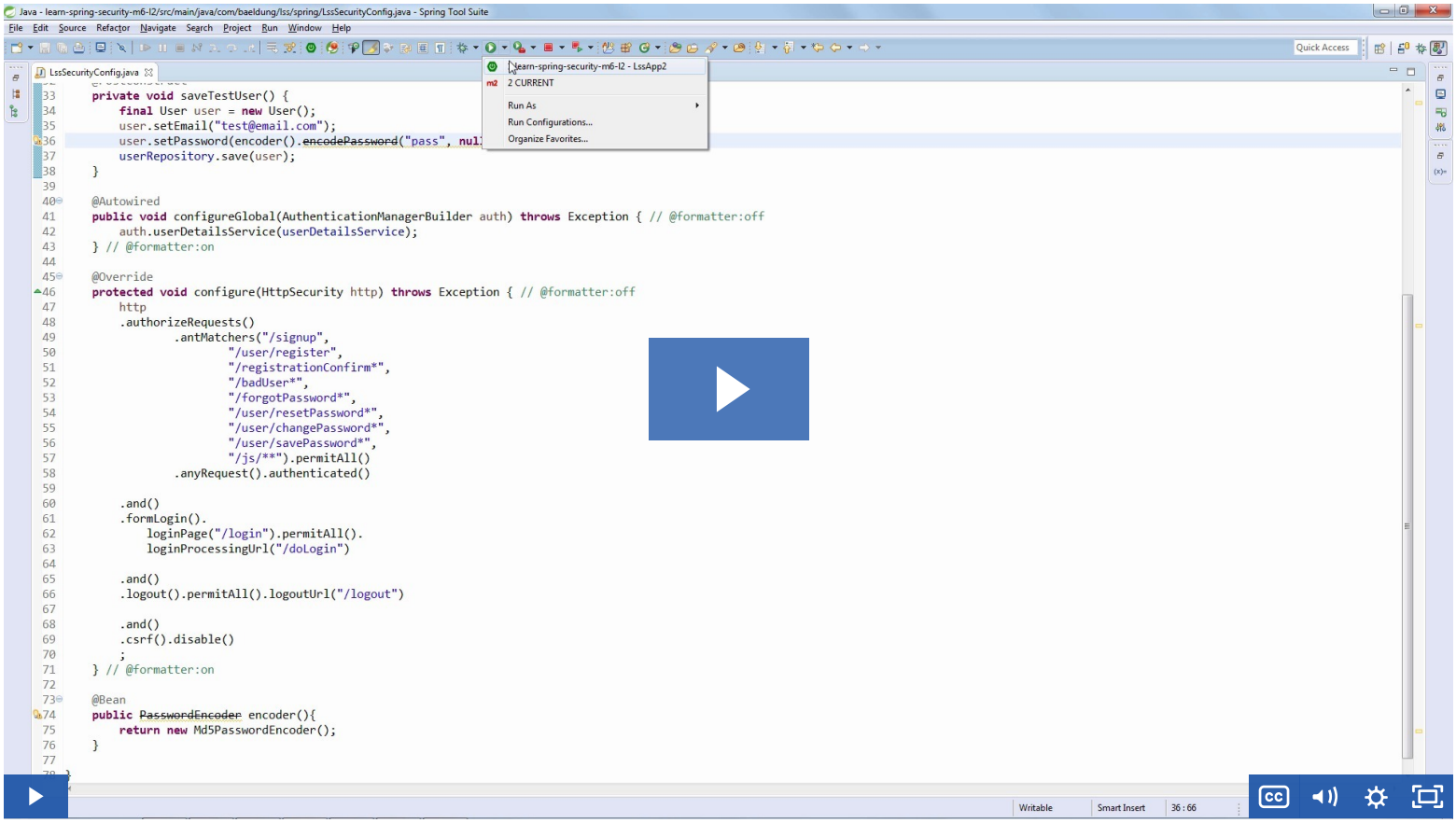
Let's establish the basics:

- we're going to store the hash the password
- we're going to use a salt that's unique for each user
- we're going to use a secure, one way algorithm
- we're going to store the result (and throw away the password)

And the basics of the authentication flow:

- we're going to apply the same one-way algo and the same salt
- and we're going to simply compare

Lesson 2: Hashing Passwords (MD5 and SHA-256)



1. Goals

The goal of this lesson is to give you an intro to hashing passwords.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m6-lesson2

If you want to skip and see the complete implementation, feel free to jump ahead and import: m6-lesson3

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

First, we'll switch back to MySQL - to be able to use the MySQL Workbench and explore how the data is stored.

If we run the project as is and we have a look at the DB, we'll see the password being stored in plain text. Let's change that.

A quick note, before using a password encoder - notice that we have 2 *PasswordEncoder* interfaces in Spring Security.

One of the is deprecated - we're going to explore that first, just to get some context around how a simpler implementation works.

2.1. First - A Deprecated Password Encoder

First, let's define the bean - we're going to use the MD5 implementation:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new Md5PasswordEncoder();
}
```

Now, let's actually wire the bean:

```
auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder())
```

And we're going to actually encode the password, when the user was created:

```
user.setPassword(passwordEncoder().encode("pass", null));
```

Finally, we're going to run the system and have a look at the DB - where we can now see the password no longer being stored as a plain-text value.

2.2. The New Password Encoder

OK, let's now switch to the standard (and non-deprecated) implementation of the encoder:

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

Note that this implementation is using SHA-256.

Finally, let's use the password encoder into our *UserService* - to make sure that passwords always get encoded properly:

```
@Autowired
private PasswordEncoder passwordEncoder;
public User registerNewUser(final User user) throws EmailExistsException {
    ...
    user.setPassword(passwordEncoder.encode(user.getPassword()));
    return userRepository.save(user);
}
public void changeUserPassword(final User user, final String password) {
    user.setPassword(passwordEncoder.encode(password));
```

```
    userRepository.save(user);
}
```

3. Resources

- Password Encoding in the Spring Security Reference

- Registration with Spring Security – Password Encoding

Lesson 3: Why Hashing Isn't Enough - Using Salts

1. Goals

The goal of this lesson is simple - moving past just hashing the password and introducing salts.

2. Lesson Notes

This lesson is theoretical and doesn't have/need accompanying code.

First, a few things to avoid/not do:

- use a single system-wide salt
- just use a salt that's unique per user

Salts should be unique per credentials - so, if the user changes their password, they get a new salt.

It's also important to understand that the security scheme doesn't depend on the salt being hidden - the salt value can and should be saved as is in the DB.

In conclusion, **a salt should be a public fixed-length cryptographically-strong random value**.

The credentials used in the code of this lesson are test@email.com/pass (PostConstruct)

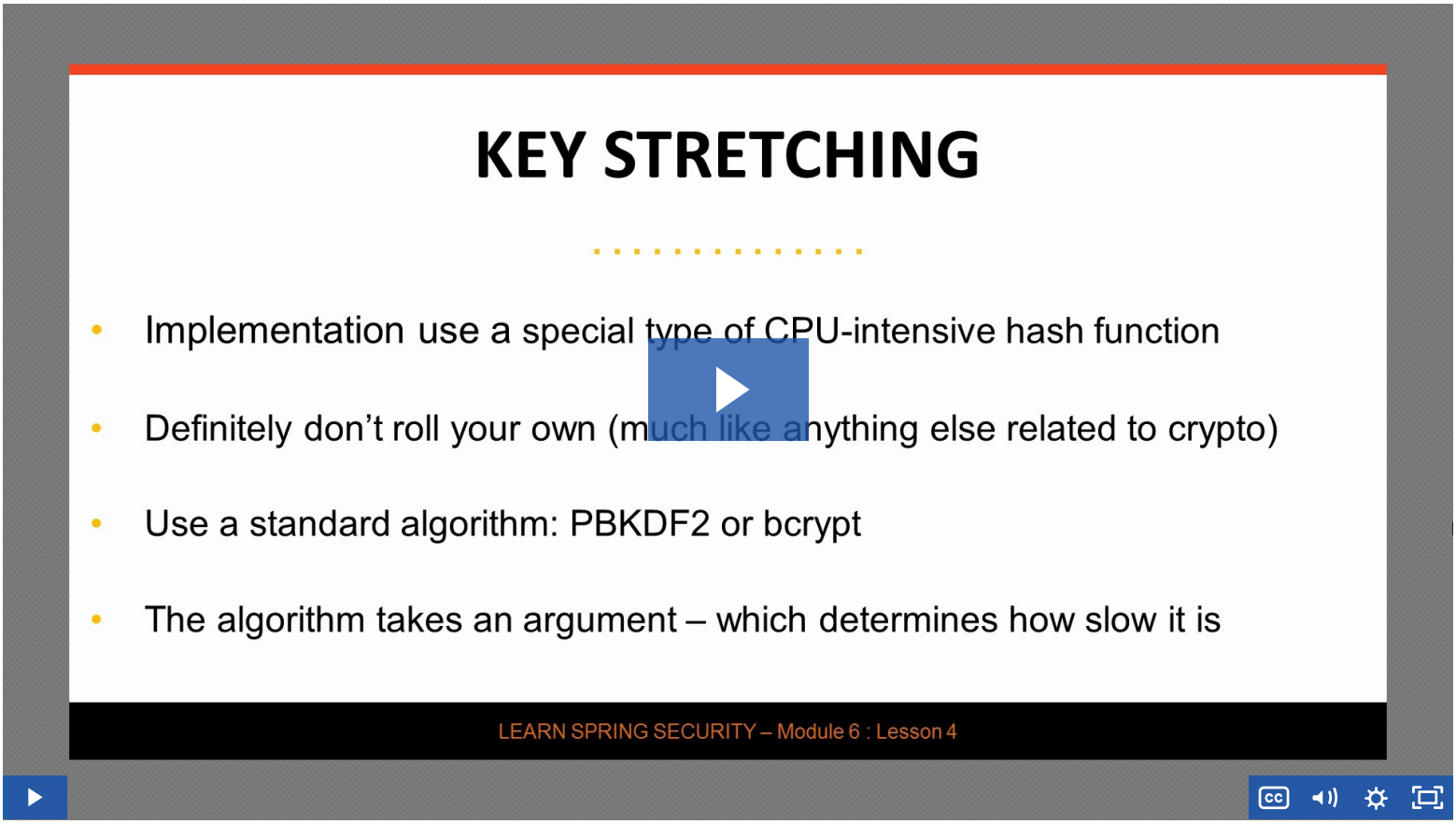
On to the implementation - with the older password encoders, we needed to define a salt source bean (for example a ReflectionSaltSource).

However, now that we're using the *StandardPasswordEncoder*, we no longer need to do that - the implementation has an internal salt generator. This is a 8-bit random salt value that's generated via a Java SecureRandom and is stored as the first 8 bytes of the hash.

3. Resources

- Adding Salt to a Hash in the Spring Security ReferenceAdding Salt to a Hash in the Spring Security Reference

Lesson 4: Key Stretching



1. Goals

The goal of this lesson is to introduce the concept and the need for key stretching when storing passwords.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m6-lesson4

If you want to skip and see the complete implementation, feel free to jump ahead and import: m6-lesson5

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

Using salts makes a large set of passwords harder to crack using specialized attacks (lookup tables, rainbow tables, etc). However, these can still be cracked by brute-forcing them individually (or by using dictionary attacks).

And of course high-end hardware (GPUs) can run/try a very high number of of hashes per second.

This is where key stretching comes in.

The goal is to make the hash function very slow.

That way attacks become impractical - but still fast enough so that the user isn't delayed to much (remember that we are running that hash function every time they log in).

The standard algorithms (PBKDF2 or bcrypt) - take a security factor or iteration count as an argument - and that determines how slow the process is.

A good goal to hit is 500ms - slow enough but it won't significantly affect user experience.

Now - it's important to understand and be aware of **the consequences** of implementing key-stretching.

First, the application will need extra computational resources to process large volumes of authentication requests.

And second, Denial of Service (DoS) attacks will be easier to mount.

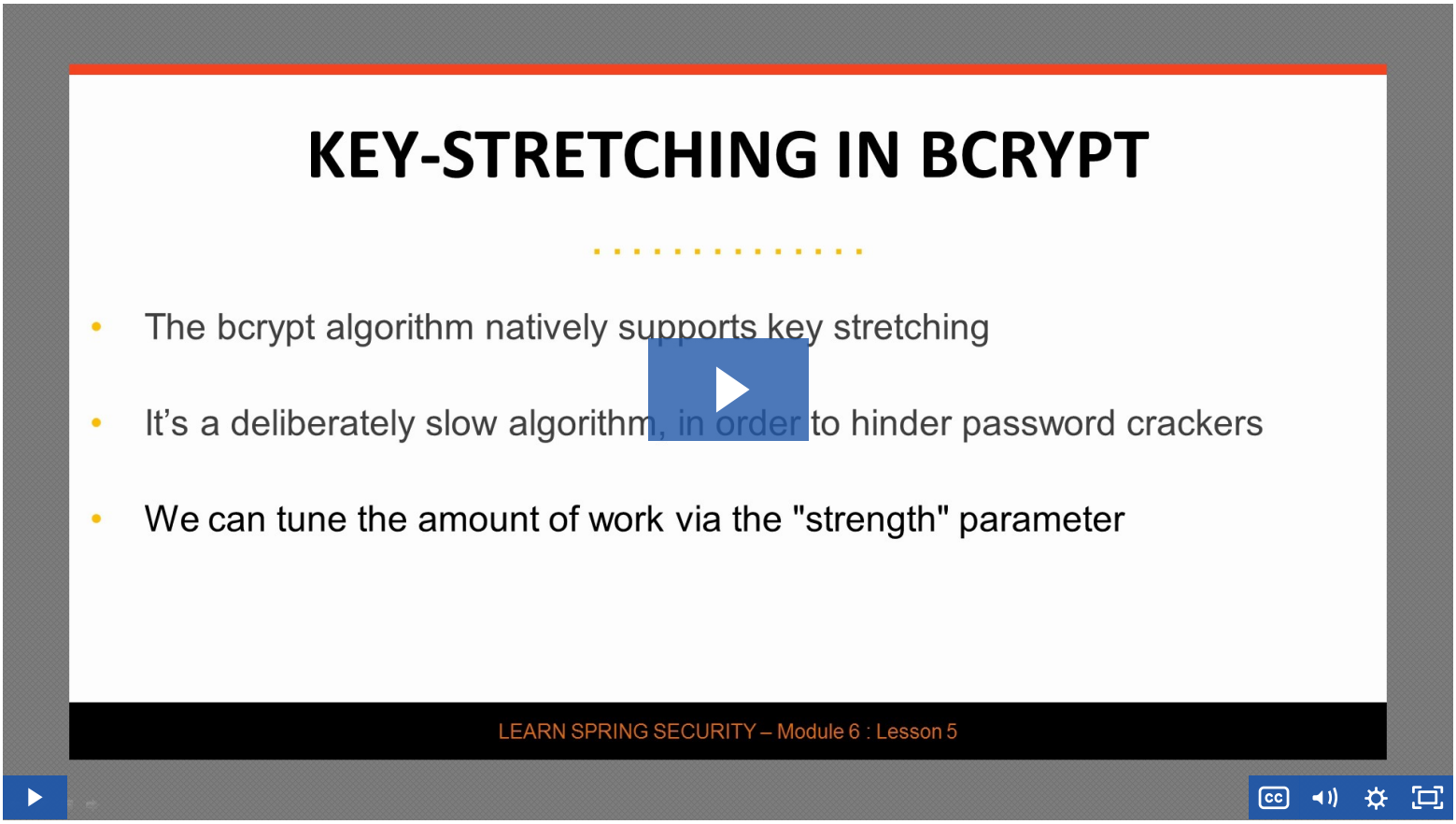
A simple way to **mitigate these risks** is to pick a lower iteration count, or to make the user solve a CAPTCHA when they log in.

But what is critically important to understand is that, even if you're using a lower iteration count, the security of the credentials is still much higher than not using key-stretching at all.

3. Resources

- Key Stretching on Wikipedia

Lesson 5: The bcrypt Solution (NEW)



1. Goals

The simple goal of this lesson is to **start using brypt** instead of the standard password encoder.

2. Lesson Notes

The relevant module you need to import when you're starting with this lesson is: m6-lesson5

If you want to skip and see the complete implementation, feel free to jump ahead and import: m6-lesson6

The credentials used in the code of this lesson are *test@email.com/pass* (PostConstruct)

bcrypt uses a built-in salt value which is different for each stored password - so we can just use the built-in support and not configure anything extra to use a salt.

Note that the salt is a random 16 byte value.

bcrypt also has support for key stretching. It is deliberately slow algorithm, in order to hinder the process of cracking passwords.

The amount of work it does can be tuned using the "strength" parameter which takes values from 4 to 31 (the default is 10). The higher the value, the more work has to be done to calculate the hash.

What's important is that **you can change this value without affecting existing passwords**, as the value is also stored in the encoded hash.

Here's a quick example of a bcrypt hash:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
```

So, we have:

- *$2a$* - indicating that this is a bcrypt hash
- *10$* - the strength
- *N9qo8uLOickgx2ZMRZoMye* - 22 characters salt
- *IjZAgcfl7p92ldGxad68LJZdL17lhWy* - 31 characters hash value

Finally, on the the implementation side of things.

We're going to replace the existing password encoder with an instance of the *BCryptPasswordEncoder* - and we're also going to pass in a non-default strength value here to tune the strength of the algorithm.

3. Resources

- Registration with Spring Security – Password Encoding