Flask Security with Hashed Passwords

13 min to complete · By Brandon Gigous

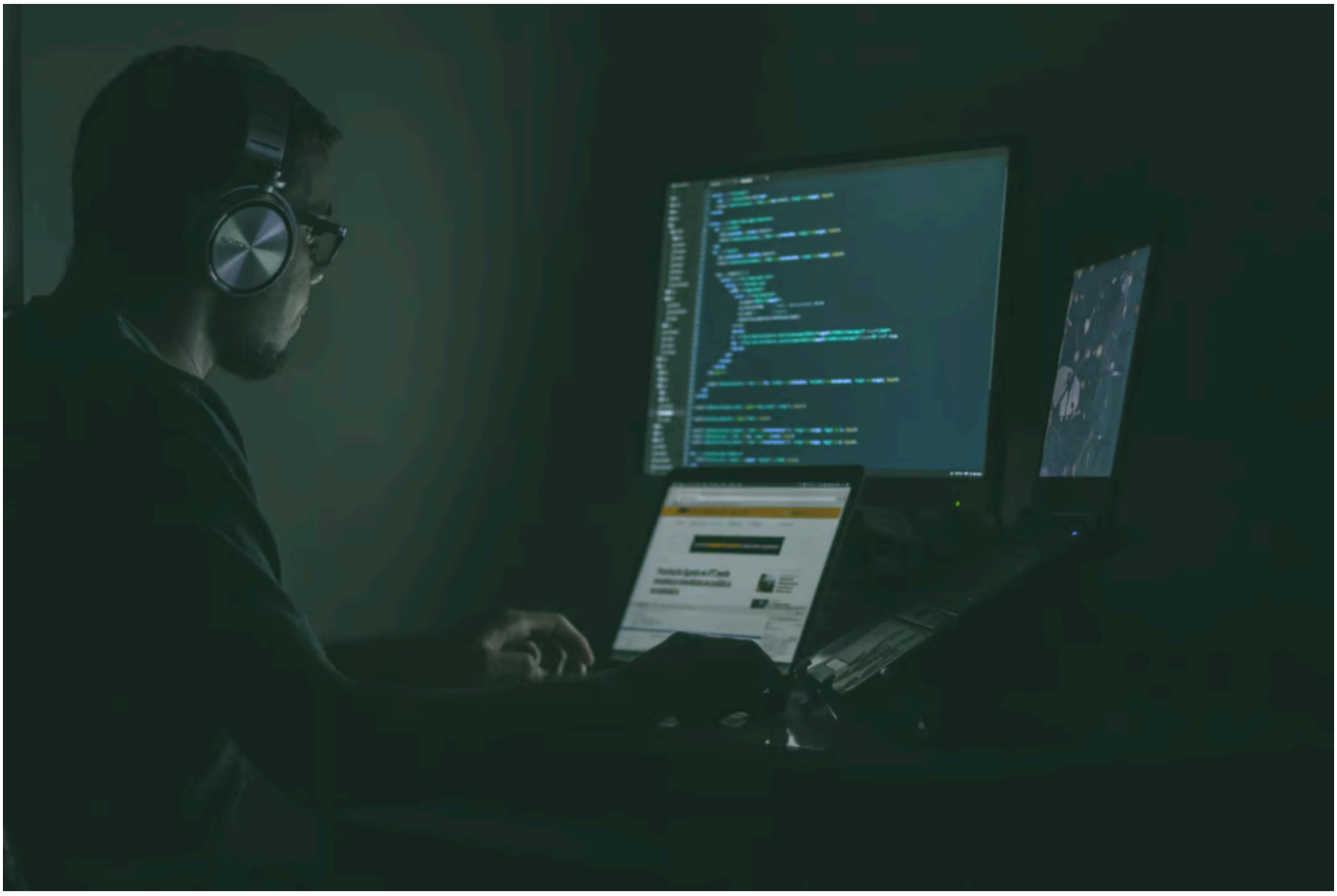
Contents

Now that you know about *why* users need to authenticate, let's move on to another important topic: **security**. Your app will be on the internet, and attackers are always lurking in the depths. This lesson will teach you the basics of **password security** by using hashed passwords.

Protecting Users' Information from Attackers

Your users are just normal people. Many of these normal people like the convenience of using the same or similar password on multiple websites. They might be people like your grandma or your quirky uncle Steve who has never gotten the hang of this "technology" thing. But *you* have better skills with technology, so you need to help them help you. That's right, help them help you because if one of your websites gets hacked, you don't all that user data to contain passwords.

Let's face it, attackers are out there. They always will be, looking to make a quick buck by stealing others information, and passwords can be a goldmine. Hackers can try those passwords on other sites and possibly steal even more information about the those users.



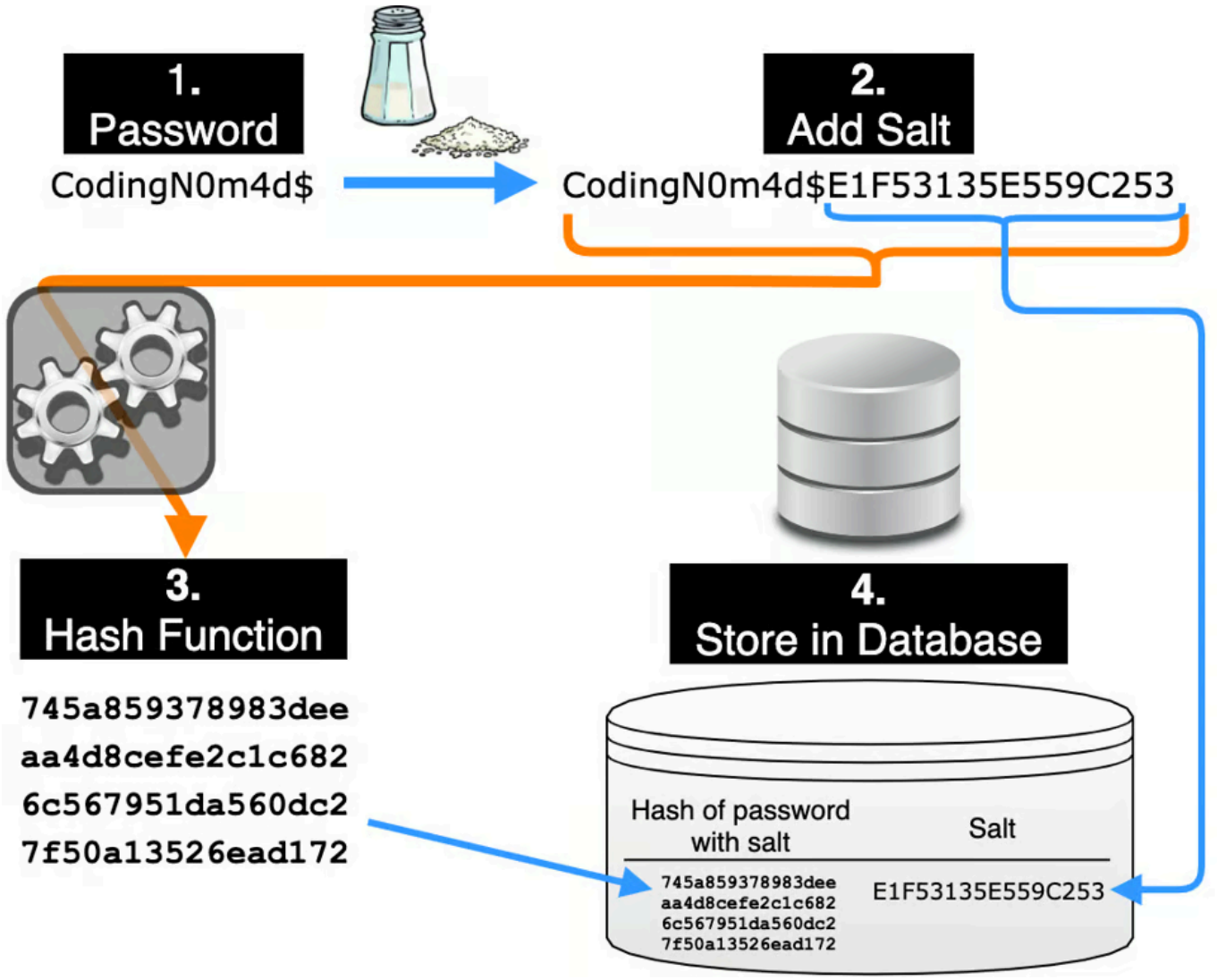
"But I have to store passwords somewhere, right?", you might be thinking. Yep, you do, but there's a way to store passwords in a database in a way that doesn't represent the actual sequence of characters needed to unlock an account.

What are Hashed Passwords?

To store passwords in a database, you can store a **hash** of it instead of the password itself. A hash is the result of taking a password, putting it though a **hashing function**, adding some **salt**, and then add some cryptographic transformations to it.

The hashing function gives something unrecognizable from the original password, but this can still be used to get the original password, and yes, hackers sometimes try to do that. The salt isn't real salt you might find on your kitchen counter top. Instead it's more digital, and can be thought of as some random component so that the hashing process isn't easy to reverse.

Recovering that password from the hash is nearly impossible, so hackers are stuck in that case. However, given the password and the salt to the same hashing function, the password can be recovered, and the user can be verified.



Hashing is cool and very complex, so it's a relief that all these computations can be done for you using state-of-the-art hashing libraries. You'll see one coming up next.

- Accountability, feedback, encouragement, and success

Get Mentorship

Give Password Hashing a Spin

To help you with all that complicated cryptography, we'll bring in the help of a library called **Werkzeug**, which is a WSGI web application library. Fun fact: "Werkzeug" actually a German word that means "tool." But that's not what you're here for, you're here to do some hashing!

Werkzeug has all you'll need in order to perform password hashing with its `security` module. It provides two important functions:

- `generate_password_hash(password, method='pbkdf2:sha256', salt_length=8)`
    - Remember all that complicated stuff about hashing? This function does all the work of taking a password in plain-text and giving you its hash version as a string. You can also change the hashing method and salt length but for your webapp, it isn't necessary.
- `check_password_hash(hash, password)`
    - This one is the reverse, in a way. Once you have a hash, you can check it against any password to see if they "match" and if they do, it returns `True`. (Hint: there's only one password that works.) So for you, you'll use this to verify that a password entered by a user is indeed the one that will let them sign in. The whole point of a *pass*word!

Adding Changes to the User Model

Note: This example builds on previous examples shown in this course.

To make a use of these functions, you can encapsulate them in your `User` model.

```
from werkzeug.security import generate_password_hash, check_password

class User(db.Model):
    # ...
    password_hash = db.Column(db.String(128))
    # ...
    @property
    def password(self):
        raise AttributeError('password is not a readable attribute')

    @password.setter
    def password(self, password):
        self.password_hash = generate_password_hash(password)

    def verify_password(self, password):
        return check_password_hash(self.password_hash, password)
```

The first thing you'll do is add the `password_hash` column. Again, this is the only "password" you need to keep around in order verify that someone's password is the correct password. As such, there's no password column and no need to even be able to access a user's password, so the `password` property throws an `AttributeError` This makes it clear to anyone with a `User` object, mainly you, that the password is a secret! However, a new password can be *set*, and once set a new password hash is calculated with the `generate_password_hash()` function.

Next is the `verify_password()` function, which uses the `check_password_hash()` function. It takes a password, likely that the user entered to login, then compares it with the hash in the database. If the two line up, it returns `True` .

Hashed Passwords with Flask Shell

What's one way you can give this a spin? If you guessed `flask shell` you'd be right! While there are other ways to try your new functionality out, all you need to do is see hashing in action. Create a new `User` and play around:

```
(env) $ flask shell
>>> u = User()
>>> u.password = 'corn'
>>> u.password
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/user/Documents/CodingNomads/projects/flask-webdev/app/
    raise AttributeError('password is not a readable attribute')
AttributeError: password is not a readable attribute
>>> u.verify_password('corn')
True
>>> u.verify_password('cork')
False
>>> u.password_hash
'pbkdf2:sha256:150000$d9qSdlno$e54fe56f649a845662a414f600791463281be
>>> u2 = User(password="tom")
>>> u2.password_hash
'pbkdf2:sha256:150000$Kfiq5Qh1$b4c922ede4424b1fb8248fccdf33a1b8a7239
```

You can see how reading the `password` attribute throws an error. You'll also notice that the password hashes from users `u` and `u2` are very different despite being based on the same password. That's due to the salting of the passwords, which adds that random component.

One way to ensure your password functionality is working is by writing a few unit tests!

Learn by Doing 

Write tests to test the following:

- That a new user with a password can be created.
- That said user has a password hash generated by your app.
- That a user's password can be verified successfully, or unsuccessfully if the password is incorrect.
- That accessing the user's `password` field results in an exception.
- That two users with the same password have different password hashes.

Don't forget to use your `new_app` test fixtures that you made in a previous lesson for these tests!

Well done, your very own password hashing and checking functionality is now complete!

Summary: Flask Security with Hashed Passwords

In this lesson you've:

- Grasped the **importance of protecting user passwords** largely due to the prevalence of people **reusing passwords** across different sites.
- Understood that while you need to store passwords, you don't have to store them in plain text. Instead, you use **hashing** to secure them.
- Learned how hashing can transform a password with a hashing function and salt, making it extremely difficult for hackers to reverse-engineer the original password.
- Discovered that **Werkzeug's security module** provides essential functions like `generate_password_hash` and `check_password_hash`.
- Implemented password hashing by adding a `password_hash` column in the `User` model and setting up methods to set and verify hashed passwords.
- Tested your password hashing implementation in Flask's interactive shell.

Great job wrapping your head around these security concepts! Now it's time to create a dedicated authentication blueprint, which you'll tackle shortly. Keep going!

Previous Next → Video: Password Hashing in Flask