Welcome back!

In this week, we will be introducing ourselves to the fundamentals of security. Broadly speaking, we are focusing on two aspects of data security- please note, that this is part-1 of a 3-part series. Over the next 2 weeks, we will encounter more security tips and tricks:

Data Security

a. Data-at-Rest
    i. Database:
        - Objective: password and secret should not be visible as plaintext
    ii. Application:
        - Objective: Should encrypt and hash the information so that its secure
b. Data-in-motion
    i. Network
        - Objective: Apply SSL certificate*

*In this tutorial, we will be applying a self-signed certificate which is unrecognized over the internet. In order to be recognized over the world wide web, we need to purchase a certificate from a trusted CA (like Comodo, GoDaddy, ssls, BigRock, Google, Microsoft, AWS etc...)

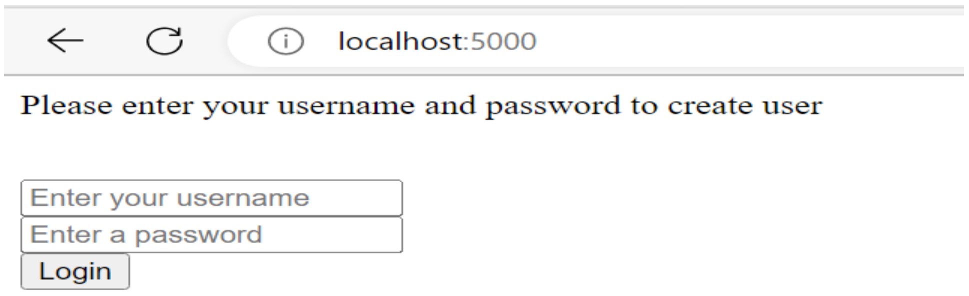
Let's begin from where we left last week. Clone the repository, get the virtual environment up and running! The URL is: [sravia-qa/QA-Assignment (github.com)]()

Once ready, run the app using:

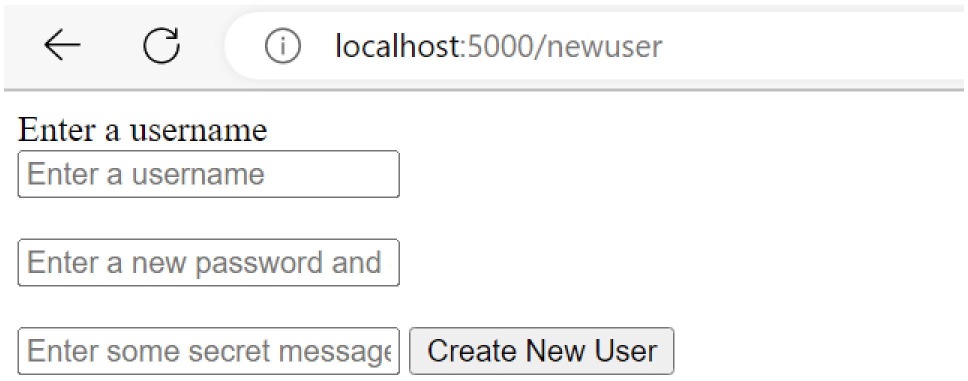
python <your app name>

You web app should then be available on your Windows' system's browser on http://localhost:5000

The screenshot for reference is below:



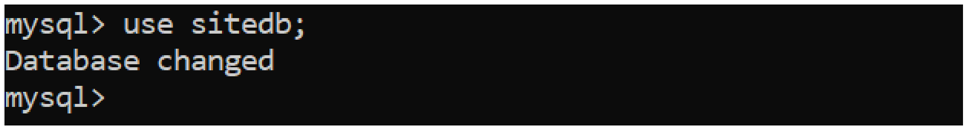
Please enter your username and password to create user



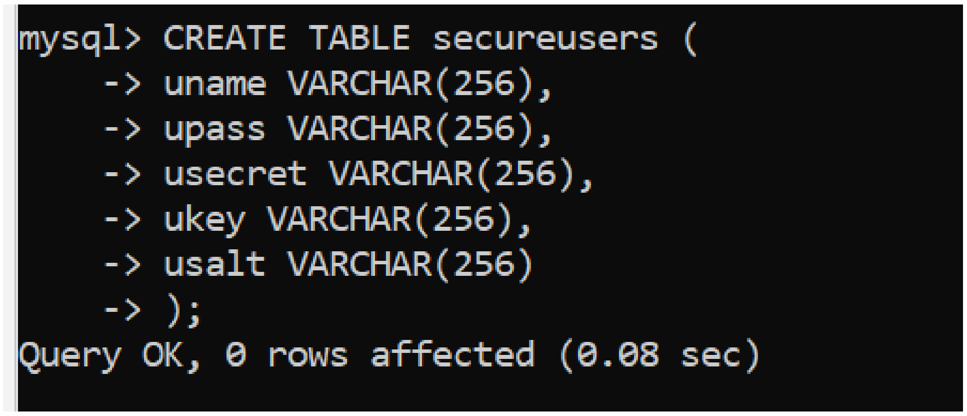
All right! Now that you have your app running, let's begin with encrypting our secret so that we can decrypt it later. However, a password is much more sensitive and should not be visible to anyone. We will perform hashing (which is one-way!) and hide that data. Before we begin, we will also need to prepare a table in our database to handle the new secure values.

Step 1: Creating a new table to store secure information

Login to MySQL command prompt, and login (steps on how to login can be referred in the previous tutorial). Now, switch to your database, and create a table as follows:



```
mysql> use sitedb;
Database changed
mysql>
```



```
mysql> CREATE TABLE secureusers (
    -> uname VARCHAR(256),
    -> upass VARCHAR(256),
    -> usecret VARCHAR(256),
    -> ukey VARCHAR(256),
    -> usalt VARCHAR(256)
    -> );
Query OK, 0 rows affected (0.08 sec)
```

Step 2: Encryption of Secret and storing the key

```
from cryptography.fernet import Fernet
```

And then, encrypt the secret:

```
usecret = request.form['usecret']

#encrypt secret

mykey = Fernet.generate_key()
```

```
encryptor = Fernet(mykey)

enc_usecret = encryptor.encrypt(usecret.encode())
```

After login, we also need to decrypt the info as follows:

```
  sql = "select usecret, ukey from secureusers WHERE uname ='"+uname+"' AND upass ='" + myhash + "'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

if len(myresult) == 0:

   return "incorrect username or password"

scrt = myresult[0][0]

mykey = myresult[0][1]

# decrypt secret

decryptor = Fernet(mykey)

dec_usecret = decryptor.decrypt(scrt)

dec_usecret = dec_usecret.decode('utf-8')

return dec_usecret
```

Step 3: Hashing of password and storing the salt

Import necessary library:

```
  import bcrypt
```

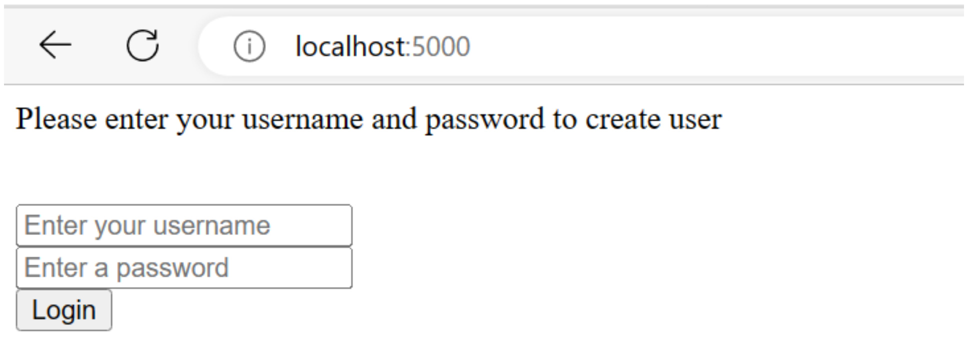
Implement hashing:

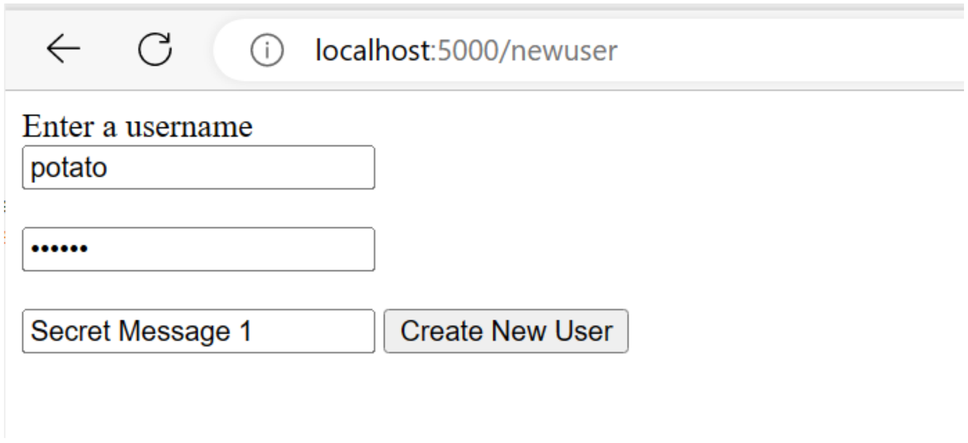
```
 byt = upass.encode('utf-8')

mysalt = bcrypt.gensalt()

myhash = bcrypt.hashpw(byt, mysalt)
```

For your reference, the end-to-end hashing and implementation is available on:

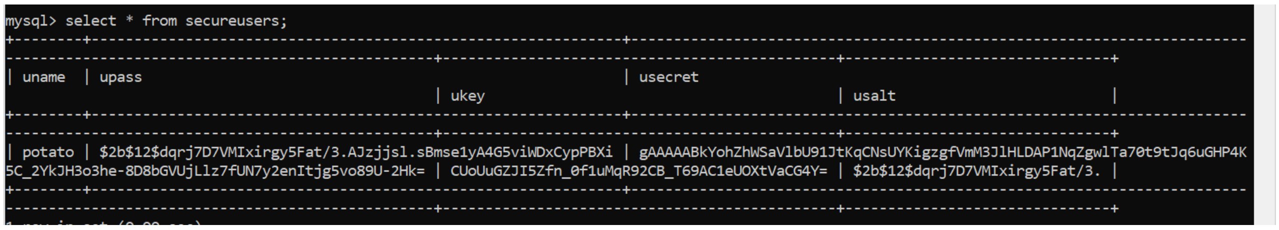
python-enc-hashing-ssl-demo/newapi_with_encryption_hashing.py at main · sravia-qa/python-enc-hashing-ssl-demo · GitHub

Now let's run the application and add a user. After the user is created, we should be able to view the secret via localhost:5000.





Now, let's observe the data in our database:



```
mysql> select * from secureusers;
+--------+--------------------------------------------------------------+-------------------------------------------------------------
------------------------------------------------+----------------------------------------------+------------------------------+
| uname  | upass                                                        | usecret
                                                | ukey                                         | usalt                        |
+--------+--------------------------------------------------------------+-------------------------------------------------------------
------------------------------------------------+----------------------------------------------+------------------------------+
| potato | $2b$12$dqrj7D7VMIxirgy5Fat/3.AJzjjsl.sBmse1yA4G5viWDxCypPBXi | gAAAAABkYohZhWSaVlbU91JtKqCNsUYKigzgfVmM3JlHLDAP1NqZgwlTa70t9tJq6uGHP4K
5C_2YkJH3o3he-8D8bGVUjLlz7fUN7y2enItjg5vo89U-2Hk= | CUoUuGZJI5Zfn_0f1uMqR92CB_T69AC1eUOXtVaCG4Y= | $2b$12$dqrj7D7VMIxirgy5Fat/3. |
+--------+--------------------------------------------------------------+-------------------------------------------------------------
------------------------------------------------+----------------------------------------------+------------------------------+
```

We have successfully now learnt how to encrypt and hash!

But what about data in motion? For data in motion, we need to ensure that our channel is encrypted. In order to do that, we need encryption keys. Alternately, we can also generate an "adhoc" key directly in our Flask API. This is a self-signed certificate, and is good enough only to build and test. We cannot use this in production- there we ned to purchase an SSL certificate from a 3rd party provider (CA).

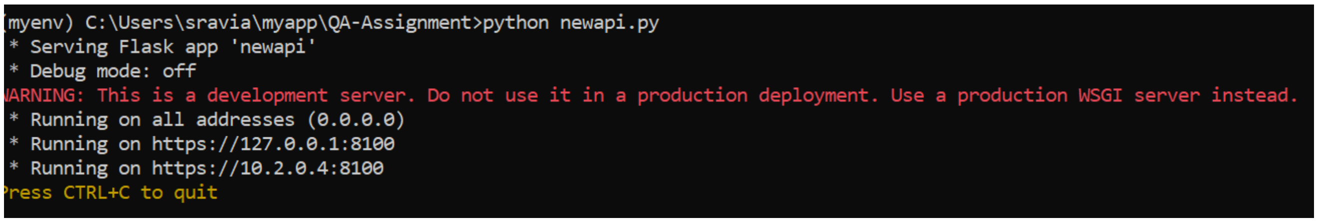
Replace app execution as follows: (By default HTTPS is on port 443, but for example we are using 8100 instead) app.run('0.0.0.0',port=8100, ssl_context='adhoc')

Complete implementation is available at:

[python-enc-hashing-ssl-demo/newapi.py at main · sraviapython -m-qa/python-enc-hashing-ssl-demo (github.com)]

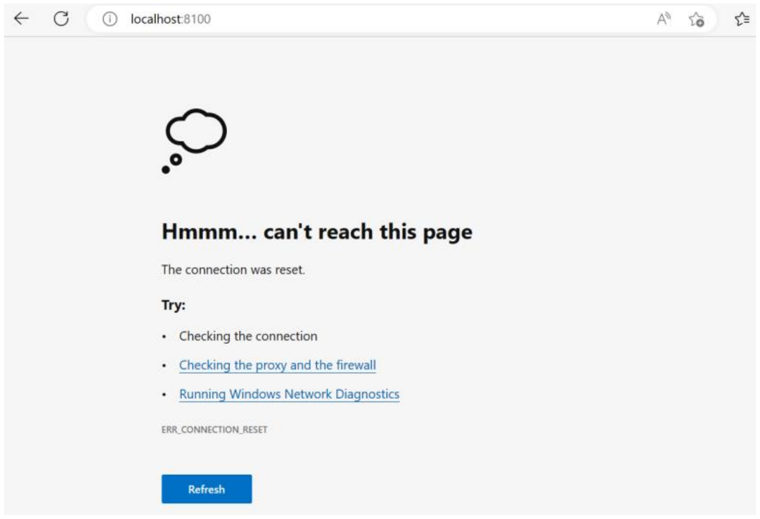
Run your python code. Observe that this time, we are running it on port 8100 and not 5000.

You may get a warning on your browser telling you not to trust the certificate. It's all right, we already know it was created by us. You can bypass the warning and access the site.



```
(myenv) C:\Users\sravia\myapp\QA-Assignment>python newapi.py
 * Serving Flask app 'newapi'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on https://127.0.0.1:8100
 * Running on https://10.2.0.4:8100
Press CTRL+C to quit
```

Verify that HTTP (insecure plaintext mode) is no longer supported:



localhost:8100

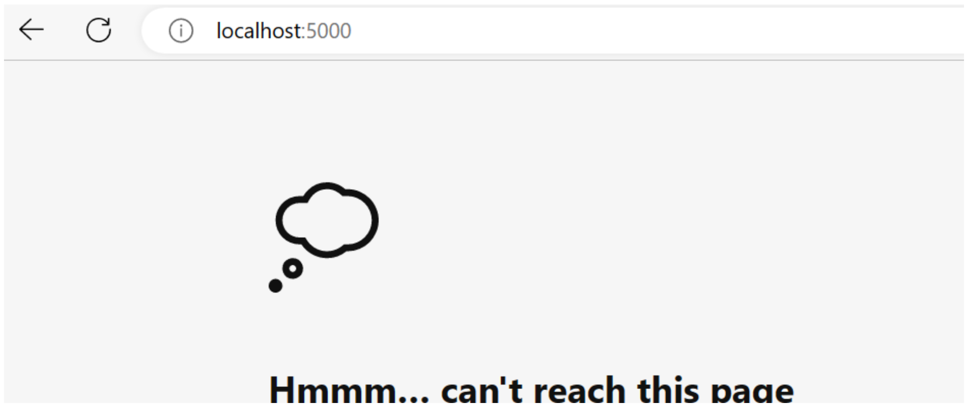
Hmmm... can't reach this page

The connection was reset.

Try:

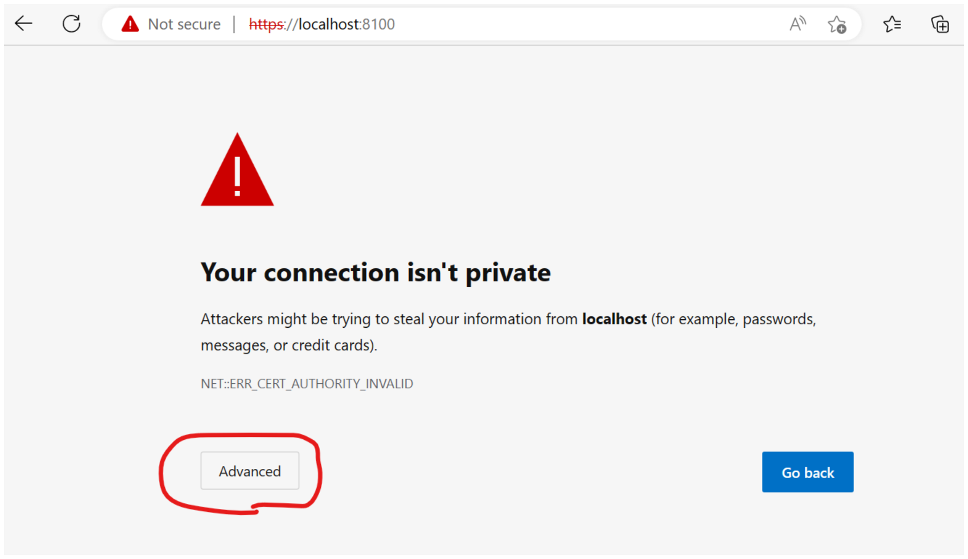
- Checking the connection
- Checking the proxy and the firewall
- Running Windows Network Diagnostics

ERR_CONNECTION_RESET


localhost:5000

Hmmm... can't reach this page

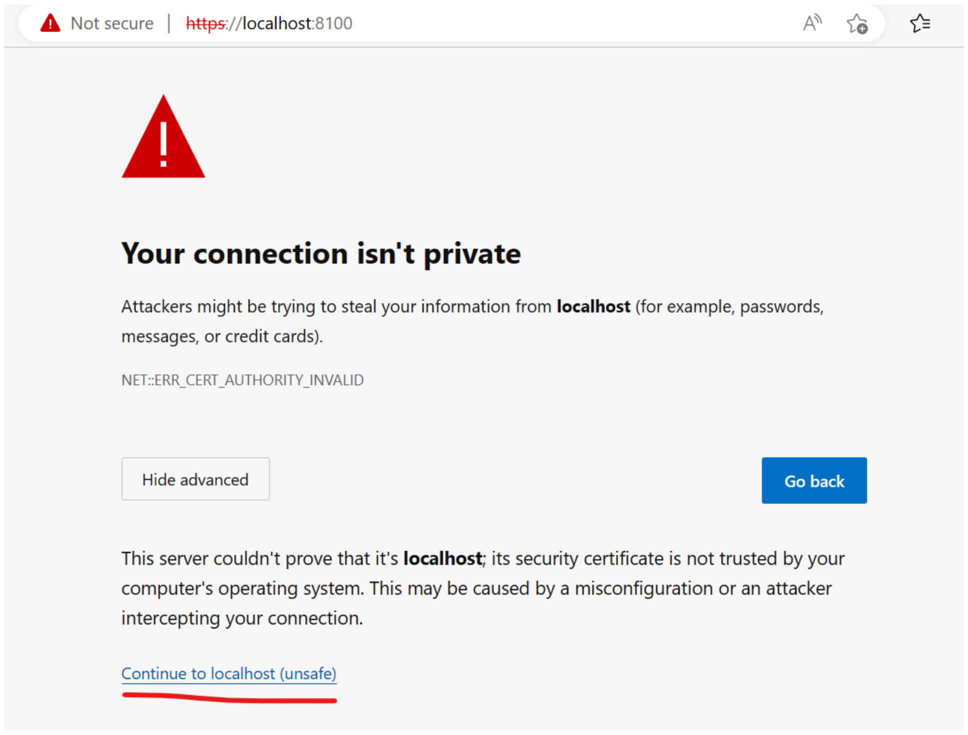
Now let’s check out HTTPS:


Not secure | ~~https~~://localhost:8100

Your connection isn't private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards).

NET::ERR_CERT_AUTHORITY_INVALID

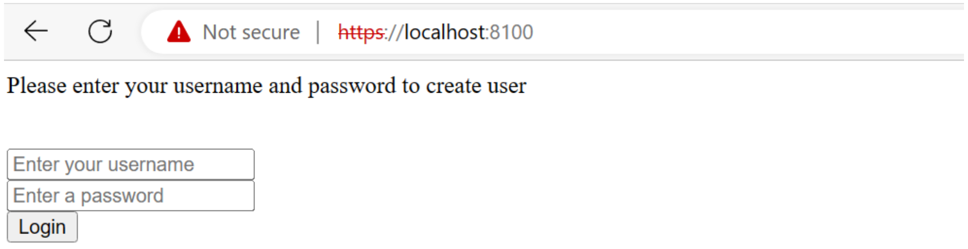

Not secure | ~~https~~://localhost:8100

Your connection isn't private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards).

NET::ERR_CERT_AUTHORITY_INVALID

This server couldn't prove that it's **localhost**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.


Not secure | ~~https~~://localhost:8100

Please enter your username and password to create user

Now we have a built a basic secure app! See you with more next week!