University of Sunderland

Technology Management for Organisations

Workshop 8

Database Security & NoSQL Applications

Aims of the workshop

Previously we have looked at creating PonyORM entities and mapping these to a relational table. This was achieved via making a new Table from scratch for an animal in a veterinarian scenario. Where previously we worked with SQL Queries directly, last workshop focused on leveraging the power of PonyORM for complex queries using generator syntax and more pythonic approaches.

In lectures we have looked at NoSQL as a catch-all term for a group of technologies primarily focused on the BASE philosophy (in contrast to ACID). Whereby, these database technologies are designed for scalability and flexibility in mind.

In this workshop we are going to look briefly at a previously ignored part of the remote database connection chain: encryption. (Lecture content to follow), as well as looking at NoSQL more in-depth with some scenarios and independent research.

Make sure to fill in your ePortfolio for Teaching Week 5 / Workshop 5 during/after the workshop.

Feel free to discuss the work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the contents of the workshop with the delivery team, and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Exercises

By now we should be able to create arbitrary Entities within PonyORM to map Python based objects to a relational database (MariaDB in our case). This is handled automatically for us. We first define our Entities, bind ourselves to a database (connect), and then generate our mappings. However, at the moment when we are connecting to our Database we are not doing this securely!

Most applications, when residing on the same server as the database, will communicate strictly over localhost - no network packets leave or enter the machine, it's all internal. As such, this affords the communication between application and database a level of security (assuming the machine is locked down from outside access / unauthorised users).

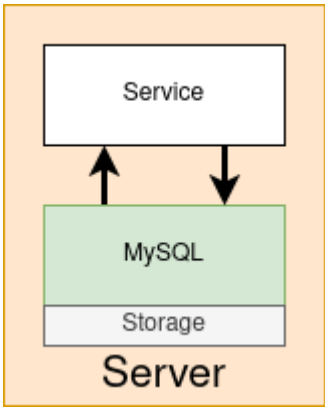

Figure 1 - Monolith, communication internal only. No external actors can read Database communications

Unfortunately, we've been running applications in the Labs/at Home (i.e Remote connections to our Server). This requires sending data over the internet, via potentially many nodes with the credentials needed to connect to the Database as well as database instructions (make tables, input data etc). This has all been unencrypted so far. See Figures 2 & 3 for detail.

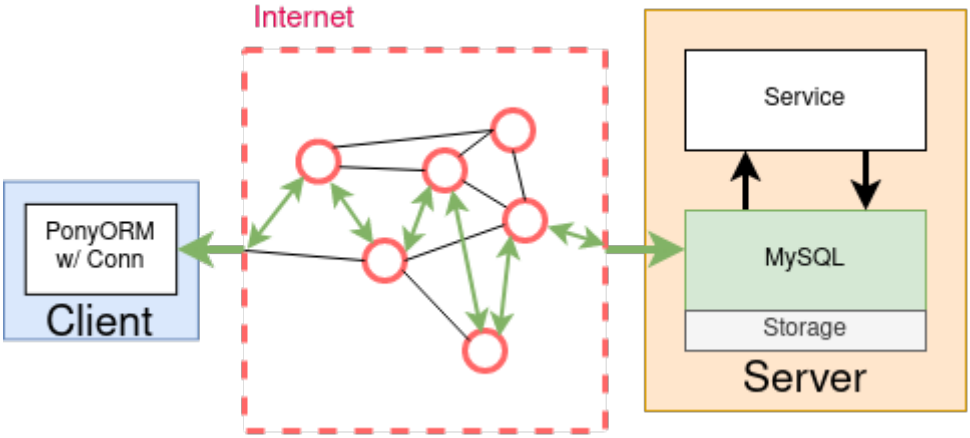

Figure 2 - Remote Connection example routed via many interconnected nodes of the internet. External actors have opportunity

With an unencrypted connection between the MySQL client (You at Lab / Home) and the server (Europa.ashley.work), someone with access to the network could watch all your traffic and inspect the data being sent or received between client and server. (Sniffing your packets)

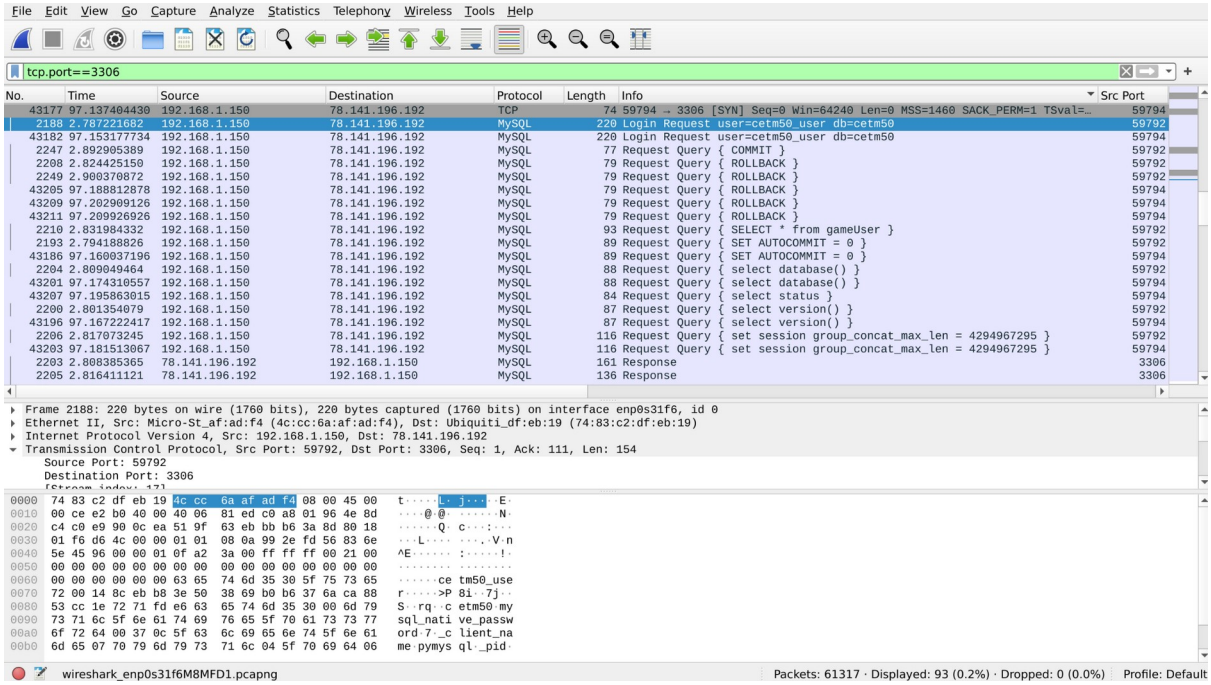

Figure 3 - Wireshark output sniffing on the MySQL 3306 Port. SQL Queries are readable by anybody who can intercept the packets. I.e Requests are in the clear.

When you must move information over a network in a secure fashion, an unencrypted connection is unacceptable. To make any kind of data unreadable, we should use encryption. Encryption algorithms must include security elements to resist many kinds of known attacks such as changing the order of encrypted messages or replaying data twice.

To secure our connection we can tell PonyORM, and the underlying MySQL Database connector, to establish a connection using TLS (Transport Layer Security). Commonly you may see this denoted as ‘SSL’ (Secure Socket Layer) within documentation / command arguments. Note: SSL is considered ‘weak’ and TLS should be used instead. TLS is more recent, and built upon SSL (See Figure 3).

Note: Our PhpMyAdmin website connections are all secure. Most of you may have noticed the ‘green padlock’ where Lets Encrypt verifies who we say we are, and that we’re using TLSv1.2. Feel free to click the padlock and ‘More Information’

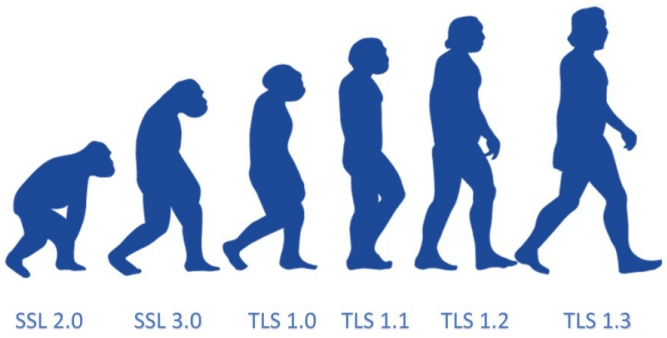

Figure 3 - Evolution of SSL / TLS

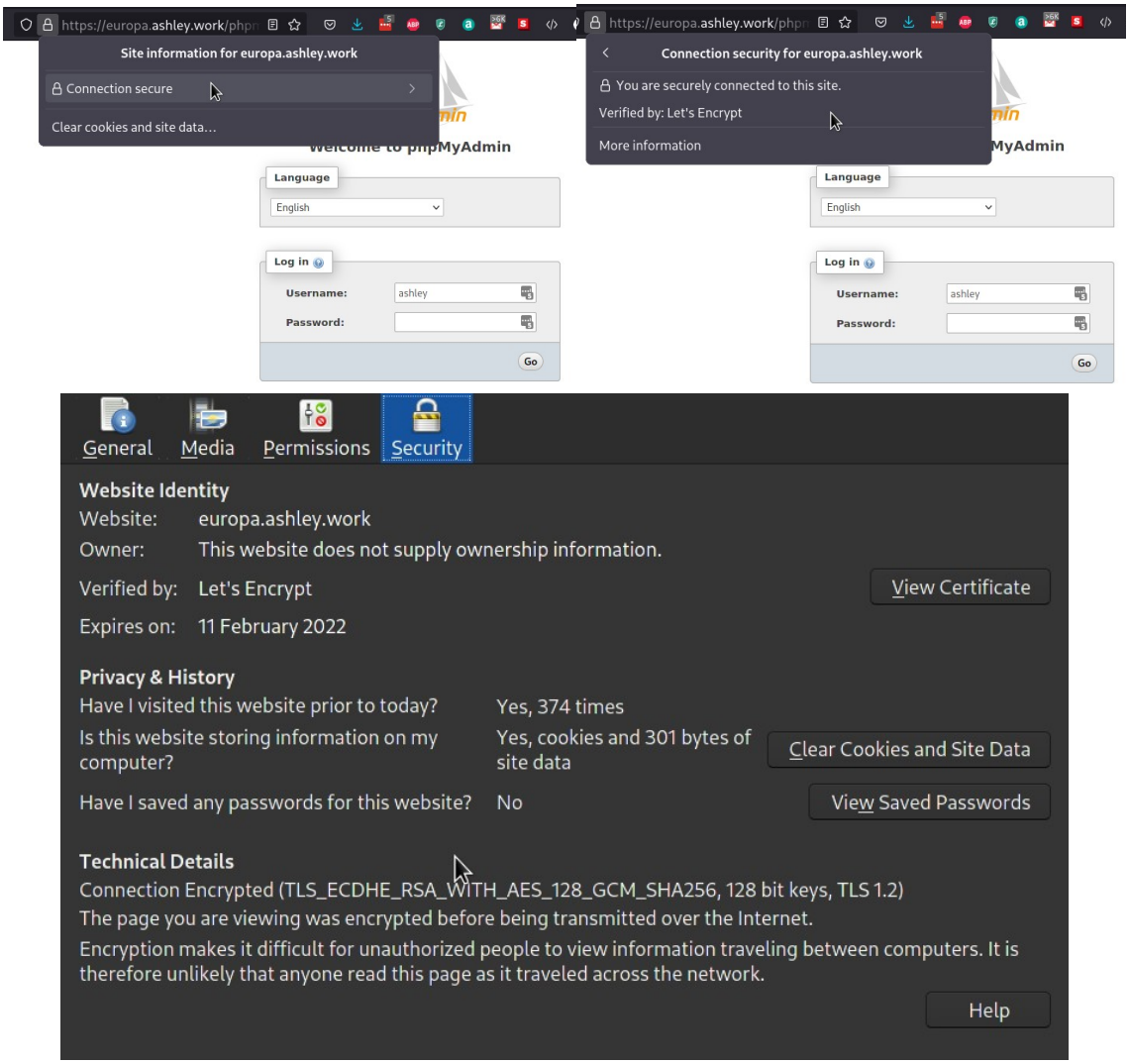

Figure 4 - HTTPS Certificate verified by Lets Encrypt using TLSv1.2 for Europa.ashley.work

Securing PonyORM

In order to secure our PonyORM connections from Python we need a few things.

1. A RDBMS which supports TLS/SSL encryption
2. RDBMS Configured to allow TLS connections
3. A MySQL Connector which supports the TLS protocol
4. A certificate to verify the server (I have provided this if your system doesn't have it)
5. Some code which will utilise this certificate, verify the server, and establish a TLS connection for encrypted traffic.

Numbers 1 and 2 have been done on the server-side already. The MySQL connectors you have been using with PonyORM support TLS arguments. Number 4 is provided on canvas as **ca-cert.pem** - Normally this will exist somewhere on your system in a default location; however, we will specify it to simplify matters. Let's look at number 5.

Previously we may have a db.bind line which looks like the following:

```
db.bind(provider='mysql', host='europa.ashley.work',
user='student_bh12xy', passwd='my_super_secure_password',
db='student_bh12xy')
```

To enable secure connections, we need to pass the underlying MySQL connector some SSL information. Namely, the certificate to verify our database against. For now, think of this as a trust exercise. **ca-cert.pem** is a way we can verify that we're talking to the correct server, and not some malicious server. **Download the ca-cert.pem from canvas and place this in the same directory as your .ipynb or .py script.**

To use that we add the **ssl_ca** argument to our bind call, and point it to the **ca-cert.pem** file:

```
db.bind(provider='mysql', host='europa.ashley.work',
user='sec_student_bh12xy', passwd='my_super_secure_password',
db='sec_student_bh12xy', ssl_ca='./ca-cert.pem')
```

As the SSL/TLS option is at the connector level, you will rarely find documentation in the ORM around SSL/TLS. Therefore, we need to look at the connector documentation if we want to know more. Either MySQL Connector (https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html) or PyMySQL (https://pymysql.readthedocs.io/en/latest/modules/connections.html)

Note: The username and database name have changed as I cannot have the same username / database for two users. An account under MySQL either Requires SSL or Not, there is no optional middle ground. Therefore, to leave your previous user still working as intended for previous workshops, I have made new user accounts for the purposes of SSL connections.

For the exercises today we are hosting a database with the following credentials:

host:	**europa.ashley.work**
username:	**sec_student_ followed by your student ID**
password:	**iE93F2@8EhM@1zhD&u9M@K**
database:	**sec_student_ followed by your student ID**

E.g sec_student_bh12xy for the username, you also have a database under this name.

Note: Last workshop I recommended leaving the password as-is for the time being. While our PhpMyAdmin sessions were all encrypted and secure via TLS 1.3, our PonyORM Python connections were unencrypted and potentially insecure. However, now we will be using SSL/TLS encryption for our connection meaning that you are free to change your password should you wish. **ONLY** for the sec_student user (the one which REQUIRES TLS connection).

Exercise 1: Replace your existing db.bind call with the new SSL secured connection call.

1. Verify the connection works
2. (Optional) Verify that the data is encrypted when compared against before, e.g via wireshark
3. Check that your new user account works in PhpMyAdmin
4. Verify the new tables / entries from your existing queries work in the database.

Update: You will be unable to log in to PhpMyAdmin with your sec_student_ based ID; it will error. The reason is that our connection to PhpMyAdmin is encrypted, but PhpMyAdmin connects to the MySQL database internally via a different connection which is incompatible with 'REQUIRE SSL'. (See Figure 5)

Therefore, your sec_student_ database and tables are now viewable from your regular student_ account on PhpMyAdmin. You will only be able to browse/view your database/tables from the insecure account. (See Figure 6)

Figure 5 - Unable to log in to PhpMyAdmin with an account which has 'REQUIRE SSL' set.

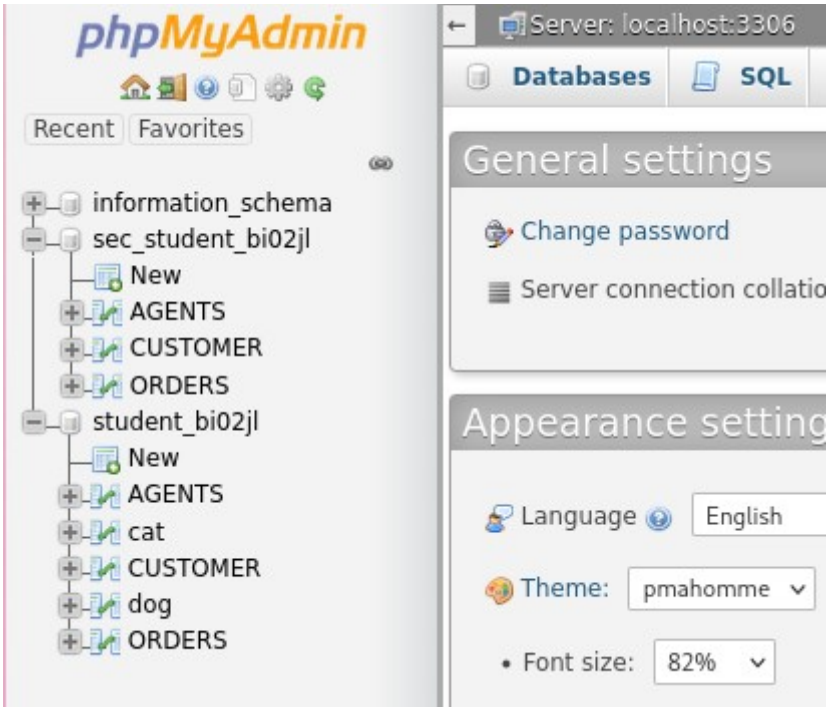

Figure 6 - Account student_ can view the student_ database and the sec_student database. Connection to PhpMyAdmin is still secure, the database user itself is not (if executing from python).

NoSQL

Exercise 2: Redis is an example of a Key-Value store which we covered in lectures, and is one of the most popular choices for this family of NoSQL Database. It is used in many new technologies, and is even open-source.

Due to it's in-memory design, it finds common use as a message-passing layer for services / distributed applications, as well as for other volatile data needs.

Visit: https://try.redis.io
Follow the interactive tutorial available from Redis. Conceptually, Key-Value stores are very different from Document stores. Key-Value works on a ***global*** key-space; everything you enter will be added to a big global table of keys (and their associated values).

Exercise 3:
In lectures we also covered a Document-store, and mentioned MongoDB as one popular NoSQL provider for this.

a) Look through the following resource, which begins by mapping Document-Store terminology to that of relational databases.
https://www.tutorialspoint.com/mongodb/mongodb_overview.htm

For both **Redis** and **MongoDB**, perform the follow set of tasks.

b) Find 2-3 use-cases of the technology being used by Medium-Large Enterprises. How are they using this technology? What for? Why did they choose it over other solutions?

c) Research how one might interact with a these NoSQL database types using Python. Compare and contrast these together briefly with how we have been interacting with our MySQL so far.

Exercise 4: Read Part VI, Chapter 15 of "NoSQL for mere mortals" (Sullivan, 2015), available on Canvas. This deals with selection of a NoSQL family, the considerations required and provides some example scenarios.
You may wish to read more about the NoSQL families covered prior to this chapter (*Hint*: This book is available in the library).

Answer the review questions at the end of the chapter (*Hint2*: We've done some of them already)

Exercise 5: As with most technology, early adoption and adoption for the sake of 'new' technology-use is a risky venture. If we implement technology within our Business without fully understanding it this can have serious ramifications and consequences - some of which are irrecoverable.

Research a use-case (separate from the below) where adoption of NoSQL led to financial ruin, or unintended consequences (data leaks, etc). You should consider what domain the company operates in, what led to the incident, the consequence of the mistake, as well as how the company recovered (if at all).

For example, in 2014 a Bitcoin exchange known as Flexcoin was attacked via a concurrency attack. The underlying finance systems of the exchange (think bank account with withdrawals and deposits but for Bitcoin) was running on a relatively new NoSQL system. As the NoSQL solution followed the BASE philosophy there were no atomicity guarantees. As such, an attacker managed to execute several rapid transactions one after another. I.e If the account balance was 10 Bitcoin, a withdrawal request was made for 9.99 bitcoin. This would pass the initial validity check, and deposit 9.99 in the target account. Unfortunately for the exchange, as ACID was not guaranteed, hundreds of these transactions were processed at once; each of which passed the initial balance check, and then deposited 9.99 in a target account. The culprit? Eventual Consistency.
The target account, and the heist, made off with 896 BTC; worth approx $500,000 at the time (now worth $32,328,397). However the initial account was only ever drained of that initial 9.99 BTC. The company could not sustain the huge loss to their holdings, and promptly folded as a company and a bitcoin exchange soon thereafter.

Exercise 6: Research performance differences between NoSQL and RDBMS Horizontal scalability. You should consider sources from academic journals, conferences, as well as any potential white papers from corporations behind these technologies; additionally you may consider third-party trustworthy empirical tests conducted.
You should discuss and compare your findings with your peers, your research work itself should be individual. Is there an outright winner in terms of performance? Or does it depend upon the application domain? (Remember to write these up in your ePortfolio).

END OF EXERCISES