Kryptering:

* Man krypterer for at skjule/sikre data, så det kun kan åbnes af en der har adgang/nøgle til at dekryptere.
* The relationship of the components that encrypt and decrypt data determine whether an encryption scheme is symmetrical or asymmetrical.
* Symmetrisk og asymmetrisk kryptering bliver ofte brugt i fællesskab, blandt andet når vi skal oprette en SSH-forbindelse.
* Kryptering involverer rigtig meget arbejde for serveren at oprette, blandt andet derfor vi lægger den over på Nginx.

Symmetrisk kryptering:

* Samme nøgle i begge ender
* Symmetrisk er med kun en nøgle hvor der kun er en kode involveret.
* Alle der skal kunne læse beskeden skal have en nøgle (Dette er dog også en svaghed, da alle så skal have denne nøgle).
* Symmetrical encryption is a type of encryption where one key can be used to encrypt messages to the opposite party, and also to decrypt the messages received from the other participant. This means that anyone who holds the key can encrypt and decrypt messages to anyone else holding the key.
* Hurtig og effektiv

Asymmetrisk kryptering:

* To nøgler involveret, en vi udleverer, public, og en vi selv beholder, private.
* Key til at krypterer en anden til at dekryptere lige som SSH.
* Her er der 2 nøgler (private og public) involveret for at kunne dekryptere beskeden. 2 nøgler på samme algoritme.
* Asymmetrical encryption is different from symmetrical encryption in that to send data in a single direction, two associated keys are needed. One of these keys is known as the **private key**, while the other is called the **public key**.
* The public key can be freely shared with any party. It is associated with its paired key, but the private key **cannot** be derived from the public key. The mathematical relationship between the public key and the private key allows the public key to encrypt messages that can only be decrypted by the private key.
* Lidt langsommere end symmetrisk kryptering
* Alle folk skal kunne kryptere med denne public key, men kun ham med den private kan dekryptere.
* Til digitale signaturer kan man vende dem om så det er dem med en public key der kan dekryptere.

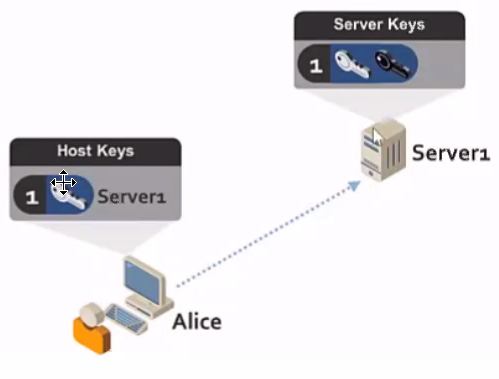
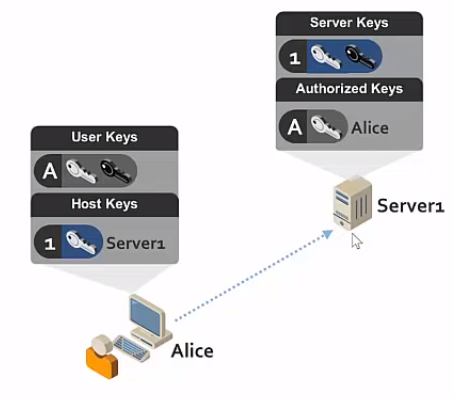
Hashing:

* Til for at mappe tekst til en fixed size.
* Hashing er en one-way algoritme, vi kan ikke komme tilbage til den originale besked, men kun checke at det vi får ud af det er det samme. Det man får ud af det kaldes et hash digest, og har typisk en fixed-size hvilket også er grunden til at vi ikke kan gå tilbage igen.
* Et hash digest/resultat skulle meget gerne være unik.
* Another form of data manipulation that SSH takes advantage of is cryptographic hashing. Cryptographic hash functions are methods of creating a succinct “signature” or summary of a set of information. Their main distinguishing attributes are that they are never meant to be reversed, they are virtually impossible to influence predictably, and they are practically unique.

Use cases:

* Kryptering over SSH
* Symmetrisk kryptering
  + Symmetric keys are used by SSH in order to encrypt the entire connection.
* Asymmetrisk kryptering
  + SSH utilizes asymmetric encryption in a few different places. During the initial key exchange process used to set up the symmetrical encryption (used to encrypt the session), asymmetrical encryption is used. In this stage, both parties produce temporary key pairs and exchange the public key in order to produce the shared secret that will be used for symmetrical encryption.
  + The more well-discussed use of asymmetrical encryption with SSH comes from SSH key-based authentication. SSH key pairs can be used to authenticate a client to a server.
* Hashing
  + Hash – salting of passwords
  + One-way hash for authenticating (Like when we installed Kali Linux)
  + Another form of data manipulation that SSH takes advantage of is cryptographic hashing. Cryptographic hash functions are methods of creating a succinct “signature” or summary of a set of information. Their main distinguishing attributes are that they are never meant to be reversed, they are virtually impossible to influence predictably, and they are practically unique.

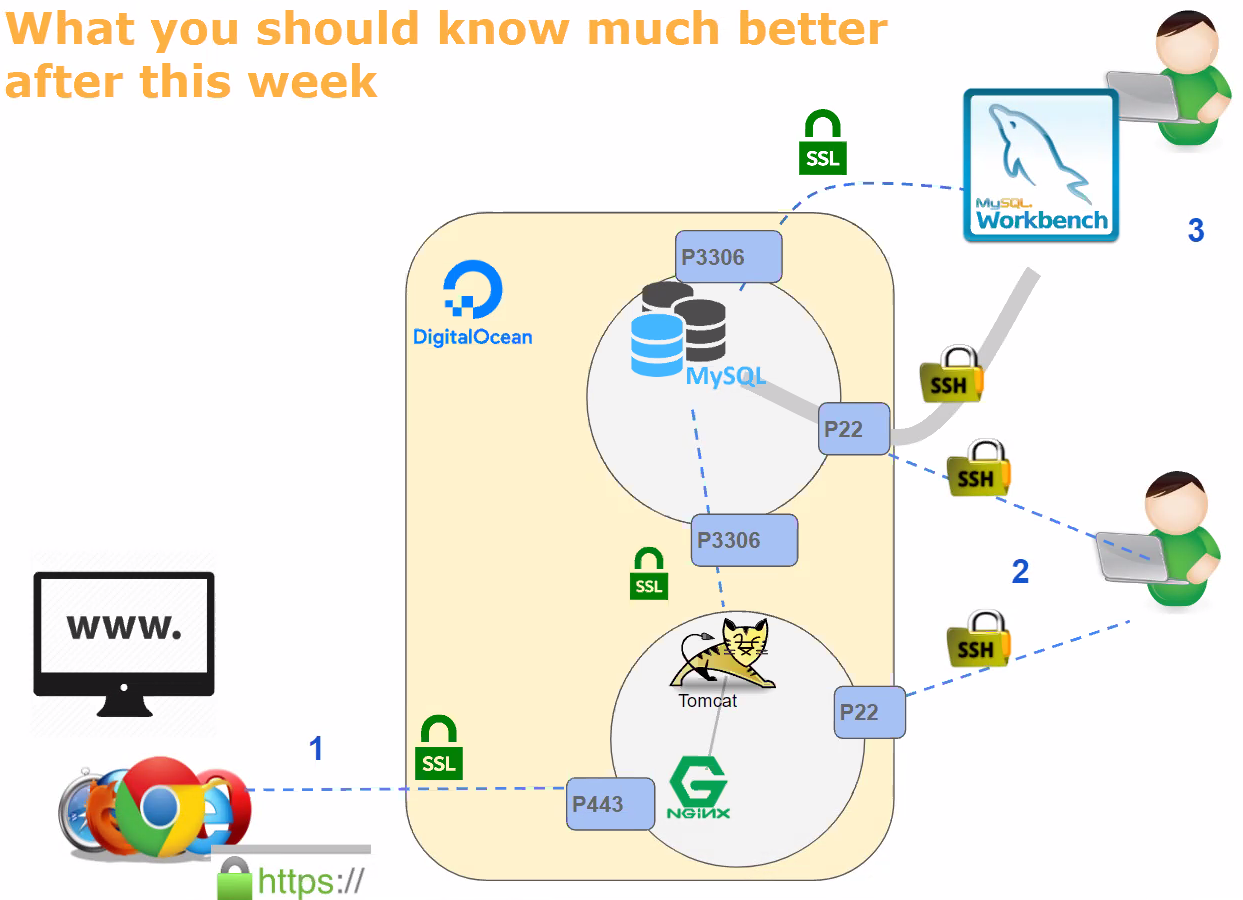
Safely log into a server using an SSH connection, without having to provide password.

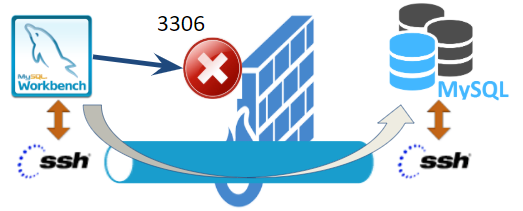
* 
* Når Alice tilgår serveren første gang gennem SSH, sender serveren en public key ud fra et key-pair som serveren genererer. Når Alice siger yes til at gemme denne public key, behøver Alice ikke at blive sendt en public key igen, men kan genkende at der tilgås den rette server ved at denne public key matcher serverens private key. Nu kan Alice og serveren kommunikere sikkert med hinanden. Dette er ikke Authentication, da alle kan få udleveret sådan en public key når de tilgår serveren første gang.
* 
* Hvis Alice så vil kunne logge ind gennem SSH, danner Alice selv et key-pair, og gemmer en public key på serveren. Nu kan Alice authenticates hos serveren der lave genererer en secret og det er kun Alice der kan dekryptere det som serveren sender da kun hun har en machene private key.
* Når vi opretter et SSH-login kan vi have en måde at lægge vores public key over på serveren. På Digital Ocean kan man checke en boks af så Digital Ocean selv indsætter denne public key in i folder der heder ’authorized\_keys’, ved oprettelsen af en droplet. Hvis man nu ikke bare kunne checke denne boks af ved oprettelsen af en droplet, skal man enten logge ind med at one-time login eller kontakte nogen der har adgang til serveren, som så manuelt kan sætte din public key ind i authroized\_keys filen. Når vi under oprettelsen af vores brugere på en ny droplet, bruger denne kommando rsync --archive --chown=**xxxx**:**xxxx** ~/.ssh /home/**xxxx** kopiere vi faktisk vores SSH-key over i brugens eget arkiv, så også denne bruger kan logge ind ved brug af SSH og vores key-pair.
* På serveren skal der altså ligge et par set af nøgler
  + Under authorized\_keys filen ligger alle vores public keys fra forskellige computere der kal kunne SSH ind på serven uden login. Det eneste formål ved at vi lægger vores public key op på serveren er at serveren kan sende os en secret som kun vi kan dekryptere da vi kar private keyen og sende denne dekrypterede secret tilbage til serveren for at konfirmere at vi er den rigtige bruger, da det kun er os der har den tilsvarende private key.
  + Under /etc/ssh/ ligger der nogle public keys indikeret med .pub og nogle private keys som serveren har genereret. Dette er dem som serveren udsender til brugere der connector for første gang under SSH. Public keys fra dette directory, er de samme som dem vi kan finde under known\_hosts på vores lokale computere, da det er dem vi har modtaget af serveren første gang vi tilgik serven ved SSH.

SSH and SSL have very strong similarities. They both offer: data-in-motion encryption, server authentication client authentication data integrity mechanisms.

Tunneling:

En forbindelse hvor vi direkte kan reagere med indhold og services på en server gennem en sikker forbindelse (SSH, secure shell) fra en anden server eller maskine. Ligesom et virtuelt rør.

SSH provides a way to secure the data traffic of any given application using port forwarding, basically tunneling any TCP/IP port over SSH. This means that the application data traffic is directed to flow inside an encrypted SSH connection SSH tunneling enables adding network security to legacy applications that do not natively support encryption.  




**Opgaver:**  
Link: [https://docs.google.com/document/d/1L3DTzTv3yR9yM3h843e1LxAmQnHX0p7-A58OJLBBgsA/edit#](https://docs.google.com/document/d/1L3DTzTv3yR9yM3h843e1LxAmQnHX0p7-A58OJLBBgsA/edit)  
  
Sæt henholdsvis Tomcat og MySQL droplet op ved at følge installationsvejledningerne og kørt de vedhæftede scripts for at installere henholdsvis Tomcat og Nginx på den ene droplet og MySQL på den anden.

Først opretter vi en database der hedder ’example’.

Herefter opretter vi en MySQL user 'no\_ssl'@'127.0.0.1' med password ’test’.

* Grunden til at vi opretter en bruger der hedder 'no\_ssl'@'127.0.0.1' er at vi opretter en bruger der kun kan begå sig lokalt på vores droptlet, men ikke kan tilgå dropletten udefra. Den hedder no\_ssl fordi vi ikke skal bruge SSL for at tilgå databasen fra denne bruger, men i stedet gør brug af SSH på en eksisterende droplet bruger og derefter tilgår MySQL lokalt, i stedet for direkte på port 3306 udefra med SSL.

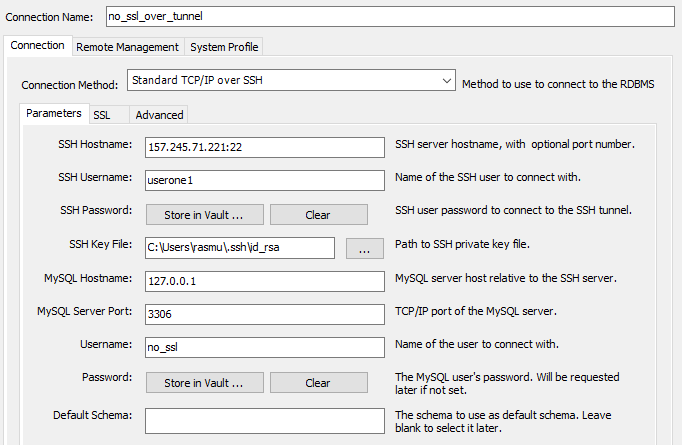
Herefter giver vi grant all rights til den nye test bruger.

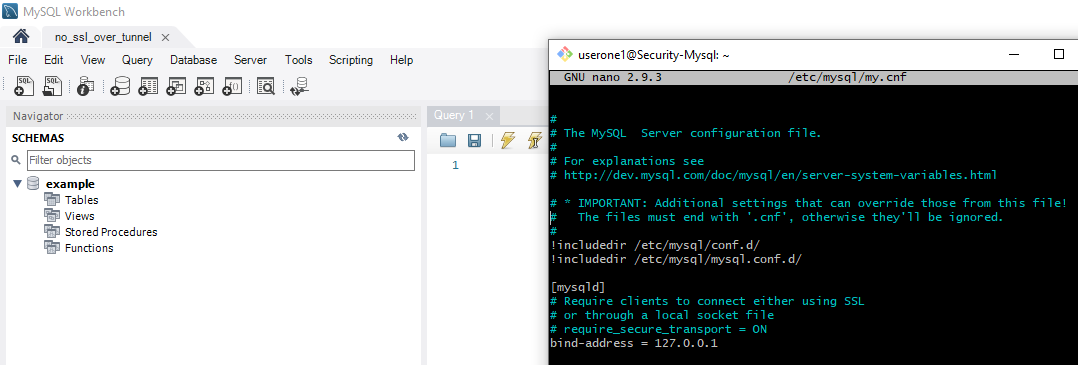
**Connecting to MySQL via an SSH Tunnel:**  
The purpose of this exercise is twofold.

It will provide a practical example of using an SSH-Tunnel and will demonstrate how we can access a MySQL server from a remote location, even if the server allows only connections from localhost, and the firewall disallows port 3306.

Herefter går vi ind i sudo nano /etc/mysql/my.cnf filen og tilføjer ’bind-address = 127.0.0.1’ under [mysql], hvilket binder MySQL til lokal host adressen på serveren.

Herefter bruger vi sudo ufw deny mysql til at lukke for adgang til MySQL på port 3306 udefra, så vi kun kan tilgå MySQL lokalt.

Efter dette opretter vi en connection på mysql Workbench. Da vi med firewallen har lukket for port 3306 til MySQL og i stedet vil bruge SSH gennem port 22 til at tilgå MySQL på 3306 lokalt, ser denne connection sådan ud:  


Her ser vi at vi kan tilgå vores MySQL database ved at forbinde over SSH på port 22 på og derefter lokalt på 127.0.0.1 localhost tilgå vores MySQL database på port 3306 med vores no\_ssl bruger.  


**Connecting directly to MySQL with Mandatory SSL**As explained [here](https://www.ssh.com/ssh/tunneling/), tunnelling comes with its own set of risks, and when we need to connect applications (a web server for example) to the database this solution is cumbersome. In this part, we will connect directly to the MySQL-server on port 3306 using SSL, both from our local developer laptop, via Workbench, from a remote Droplet and from a Java Application. The following is a customized/shortened version of [this tutorial](https://www.digitalocean.com/community/tutorials/how-to-configure-ssl-tls-for-mysql-on-ubuntu-16-04).

Change bind-addres to 0.0.0.0 så alle interfaces er tilladte.

Udkommenter ‘require secure transport’.

Vi vil gerne tillade at man kan tilgå med en TLS/SSL-forbindelse.

Herefter skal vi starte vores MySQL server op så den kun kan tilgås localhost.

Herefter laver vi en bruger der kun kan benyttes når vi bruger SSL.

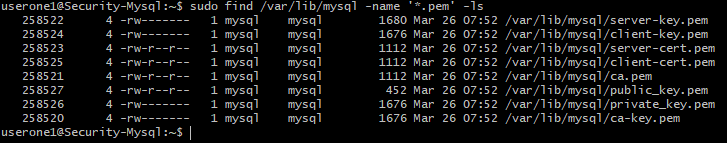
Denne bruger skal kunne tilgå vores database gennem SSL.

Herefter kan vi begrænse vores MySQL droplet gennem firewallen, så man kun kan komme ind gennem SSL fra vores anden Tomcat droplet.

* Herved kræver vi at der skal udveksles certifikater hvis maskinerne skal arbejde sammen, hvilket gør det ret sikkert.

For at kunne tilgå vores MySQL server direkte med SSL, starter vi med at ændre på sudo nano /etc/mysql/my.cnf filen hvor vi sætter værdien bind-address = 0.0.0.0. Dette vil nu tillade at vores MySQL server kan tilgås gennem remote controll. Herefter unkommentere vi require\_secure\_transport = ON i samme fil, hvilket går at kommunikation med MySQL serveren kun kan forgå over sikre forbindelser som SSL. Som det sidste køre vi kommandoen sudo ufw allow mysql så vi nu åbner for kommunikation til port 3306 udefra, da der er her vores MySQL server lytter.

Generating keys:   
You can skip this step since it seems like MySQL 8.x creates the required certificates and keys during installation. It could be argued however, that it's always best to generate these things by yourself, but I suggest you skip this section.

Vi kan checke for eksisterende certificater og keys autogenereret af MySQL og eventuelt os selv, ved at køre kommandoen: sudo find /var/lib/mysql -name '\*.pem' -ls . This should, among several other things, show the generated files with the user and group ownership.  


These files are the key and certificate pairs for the:

* Certificate authority (starting with "ca")
* MySQL server process (starting with "server")
* MySQL clients (starting with "client").

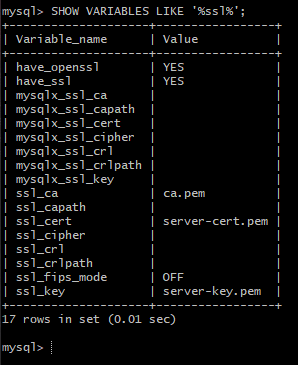
Additionally, the private\_key.pem and public\_key.pem files are used by MySQL to securely transfer password when not using SSL.

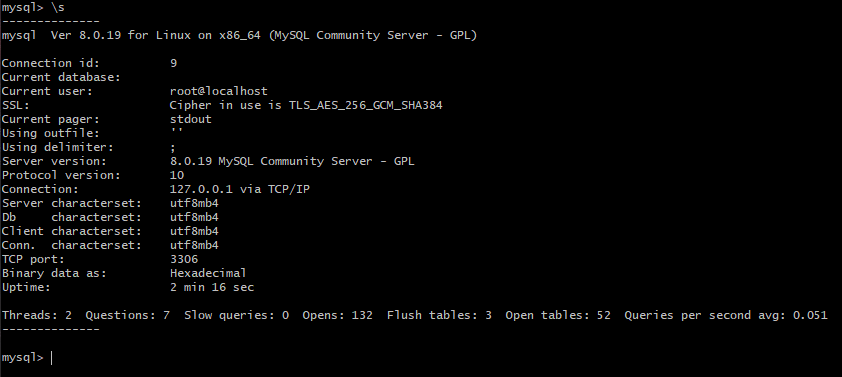
**Enable SSL Connections on the MySQL Server**

Newer MySQL versions will look for the appropriate certificate files within the MySQL data directory when the server starts. Because of this, all you need to do to enable SSL after having updated the /etc/mysql/my.cnf file is to restart the MySQL service.

Log into a MySQL session again, using the root MySQL user. THIS TIME however, to ensure that the client makes a TCP/IP connection to the local server, use --host or -h to specify a hostname value of 127.0.0.1

Take a look at the same information requested last time. Check the values of the SSL related variables:

mysql> SHOW VARIABLES LIKE '%ssl%';  


Next, check the connection details again: mysql> \s  


Ved at kigge på SSL variables på serverens lokale MySQL session, kan vi både se nogle af de keys vi har liggende til henholdsvis server keys og server certifikater, men også nogle af de cipher filer der bruges til at konfigurerer SSL for vores MySQL server i forbindelse med remote connections.

\s kommandoen på serverens lokale MySQL session, kan vi se mere om connection status og blandt andet se hvisken cipher der bliver brugt. Dette indikerer at SSL bliver brugt på vores sikre forbindelse.

Efter at have ændret vores sat vores MySQL server op til at kommunikere direkte over SSL ved remote access, kan vi oprette en ny bruger som vi kan teste vores sikre connection fra.  
mysql> CREATE USER 'everywhere'@'%' identified by 'test' REQUIRE SSL;

mysql> GRANT ALL ON example.\* TO 'everywhere'@'%';

mysql> FLUSH PRIVILEGES;

Now. your MySQL server is configured with an SSL certificate signed by a locally generated certificate authority (CA). The server's certificate and key pair are enough to provide encryption for incoming connections which we will verify in the next section.

**Testing Remote SSL Connections**

**1) From Workbench**

Vi kan teste vores SSL beskyttede connection til MySQL serveren på samme måde som vi gjorde sidst, ved at oprette en forbindelse i MySQL Workbench.

Denne forbindelse vil være lidt anderledes end sidst da vi ikke ønsker at bruge SSH til at til dropletten på port 22 og derefter tilgå MySQL lokalt på dropletten, men i stedet ænser at kunne tilgå MySQL serveren udefra gennem port 3306 over en beskyttet SSL-forbindelse.

Vi vil kalde denne connection for ’ssl\_everywhere’, da den skal kunne tilgås gennem SSL udefra.

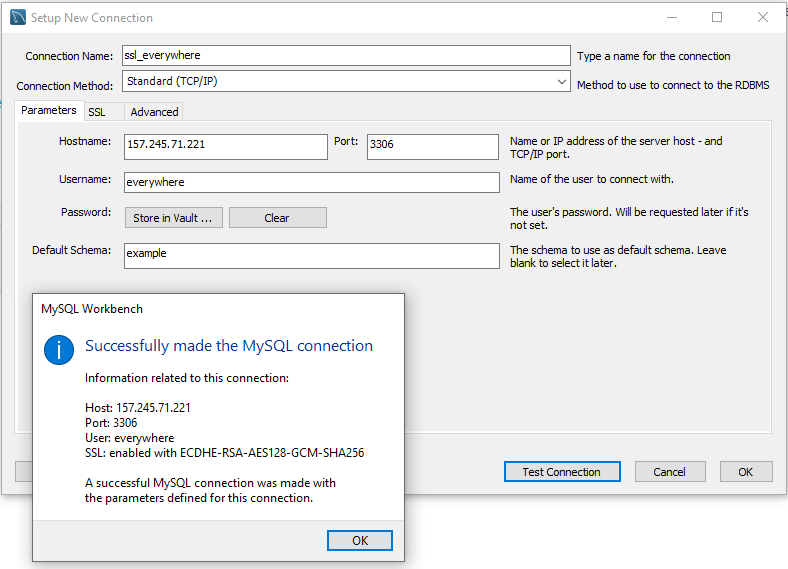
Til at starte med vælger vi “Standard (TCP/IP)” som Connection Method

Herefter vælger vi port 3306 da vi nu har åbnet op for remote access over SSL på denne port.

Herefter skal brugernavn ’everywhere’ og password ’test’ fra den MySQL bruger som vi tidligere oprettede, sættes.

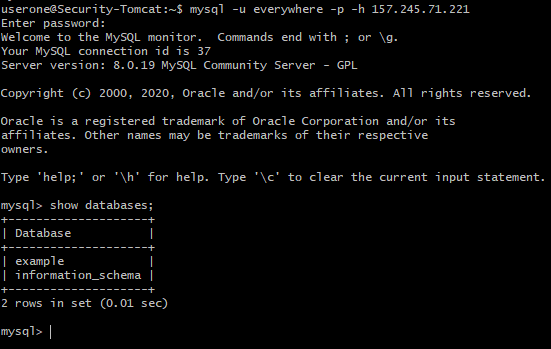
Default schema skal vælges for denne forbindelse, og det vi li dette tilfælde være vores ’example’ database, hvilket er den database som vi gav vores ’everywhere’ bruger har fået rettigheder til tidligere.

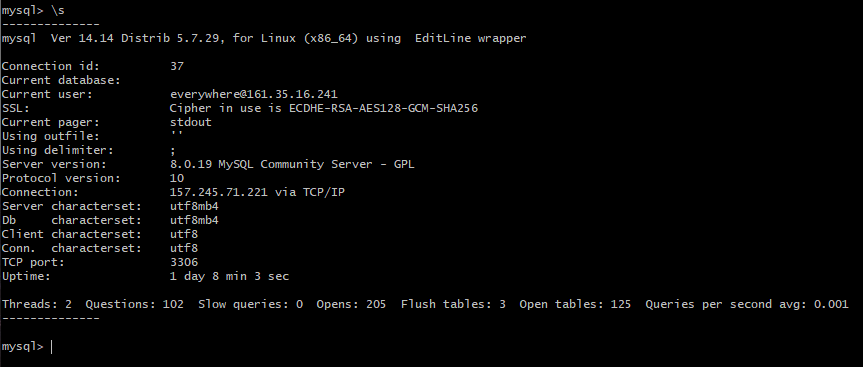
Sidst skal vi sørge for at gå til SSL taben og sætte Use SSL til ’Require’, da vi har konfigureret /etc/mysql/my.cnf filen til kun at tage imod sikre forbindelser, heriblandt SSL.

Ved at teste forbindelsen kan vi verificere at vi kan tilgå vores MySQL server på MySQL Dropletten over en sikker SSL forbindelse udefra.  


**2) From your Client Droplet (The one with Tomcat)**

For this part, you need Security-Tomcat Droplet if you already have a ready to use droplet with Nginx and Tomcat use this. On this droplet you must install the MySQL package sudo apt install mysql-client, and make sure that you can use it to connect to the MySQL droplet mysql -u everywhere -p -h **your\_mysql\_server\_IP**.



Vi kan verificere om vores connection er SSL sikker ved at køre kommandoen \s og se på forbindelseskonfigurationen. Her kan vi se forbindelsens SSL cipher, hvilket fortæller os at vi er på en sikker forbindelse med MySQL databasen på vores MySQL droplet.  


Hvis vi prøver at tilgå databasen på vores MySQL droplet med en usikker forbindelse uden SSL mysql -u everywhere -p -h **your\_mysql\_server\_IP** --ssl-mode=disabled, vil vi se at dette ikke er muligt. Grunden til dette er at vi i konfigurationen af vores MySQL server /etc/mysql/my.cnf kræver at kommunikation skal forgå over sikre forbindelser.   


**Restricting clients allowed to connect**

We can do even more to increase security for our database server

What we have done so far, is to allow access from everywhere, which is never a good idea

A quick way to restrict access to only selected IP's is via the firewall on the database server, as sketched below with this command sudo ufw allow from **SECURITY\_TOMCAT\_IP** to any port 3306.

If you still want to connect from your own laptop, you need to open up the firewall for your current IP in a similar way. Alternatively, as you have seen, you can just connect via the SSH-tunnel created in an earlier step.

**3) From a Remote Java Application**

Create a simple (non-web) Java Maven Application and repeat the following steps which should all be familiar after last semester.

* In NetBeans create a new Maven Java Application (not WEB)
* Add the  mysql-connector-java connector to the POM-file.
* Create a new Entity Class called Demo. DO NOT select “create persistence Unit”
* Add a single String-field to the class called info. Provide getters, setters and constructors as needed.
* Create a new folder **META\_INF** in  \src\main\resources
* In this folder, create a file persistence.xml and copy the content below into the file (change relevant values)

Herved kan vi lave et lille Java Maven applikation der kan forbinde over SSL til vores server og gemme 3 værdier i ’example’ databasen under brugeren ’everywhere’, hvis vores maskines IP .er blevet tilladt med denne kommando sudo ufw allow from **SECURITY\_TOMCAT\_IP** to any port 3306. Dette program kunne vi i princippet køre både lokalt og på vores tomcat server, hvorfra vi ville kunne tilgå databasen må MySQL dropletten remote over en sikker SSL-forbindelse.  
