SQL

Logging In

mysql -u root -p	how to log into mysql server on mac and assuming username is 'root'
help	gives list of possible commands.
help <category>	gives list of more specific commands
select hostname;	shows user/host name

Working with Databases

- Commands do not have to be capitalized but it is good practice from my understanding
- Commands end with a semicolon

show databases;	shows current databases
CREATE DATABASE <name>;	creates database with name
DROP DATABASE <name>;	drops/removes database with that name
USE <database name>;	to start using a database
SELECT database();	to see current running database

Working with Tables

varchar(size);	create a string of size 'size' between 1-255 characters
CREATE TABLE <table name> (column_name data_type, column_name data_type NOT NULL);	- create a table - after data_type you can specify if you want it to be NOT NULL
SHOW TABLES;	shows the names of the tables currently in the database
SHOW COLUMNS FROM <table name>;	shows the columns of the table with the name of the columns and data type
DESC <table name>;	same as above in this context but will have greater meaning down the road
DROP TABLE <table name>;	deletes the table
INSERT INTO table_name (column_name) VALUES (data);	inserts some data into a specific table
SELECT * FROM table_name;	shows all data in a table

INSERT INTO table_name(column_name, column_name) VALUES (value, value), (value, value), (value, value);	inserts multiple pieces of data into a table
SHOW WARNINGS;	shows the warnings that might pop up
CREATE TABLE <table name> (column_name data_type NOT NULL DEFAULT 'n/a', column_name data_type NOT NULL DEFAULT 99);	creates a table, but you can specify if a column should not be null and a default value
CREATE TABLE <table name>(dog_id INT NOT NULL AUTO_INCREMENT, name VARCHAR(100), PRIMARY KEY (dog_id));	- create a table with an auto incrementing primary key - can also specify 'PRIMARY KEY' at the end of the second line

Working with CRUD

- CRUD → create, read, update, delete

SELECT * FROM <table name>;	shows all columns and info from a table
SELECT <col 1>, <col 2> FROM <table name>;	shows all info in column 1 and column 2
SELECT * FROM <table name> WHERE <some clause>;	shows all columns from a table where some clause is true
SELECT <col 1> AS <alias name> FROM <table>;	shows all info in column 1, but the name of column displayed will be the alias
UPDATE <table name> SET <column>=<newdata> WHERE <clause>;	updates column in table where clause is true
UPDATE <table name> SET <col 1>=<data1>, <col 2>=<data2> WHERE <clause>;	showing that you can update multiple columns at once in the SET
DELETE FROM <table> WHERE <clause>;	delete from table where clause is true

Sourcing in .sql Files

- make sure you are in the folder where the file is before starting sql server
- start sql and select database

source <filename>.sql;	- this will run the command listed in the file - e.g. create a table with some starter data

Other Data Manipulation - will not affect data just how it's displayed

CONCAT

CONCAT(x, y, z);	used to combine columns x, y, and z for displaying
CONCAT(x, '~', y);	- used to combine columns x and y with a squiggly line between them for displaying
CONCAT_WS('<separator>', x, y, z…);	this is CONCAT with a separator, and will add the specified separator between all columns

SUBSTRING

SELECT SUBSTRING('Hello World', 3);	prints out all chars after the third one (first 'l')
SELECT SUBSTRING('Hello World', -3);	prints out last 3 char
SELECT SUBSTR(<column>, starting, ending) FROM <table>;	prints out between starting and ending point of column

REPLACE & REVERSE

SELECT REPLACE('Hello World', 'Hell', 'H***');	replaces word 'Hell' with 'H***'
SELECT REPLACE(<column>, 'e', '3') FROM <table>;	replaces all e's in column with 3's
REVERSE('DOG');	will return 'GOD'

CHAR_LENGTH, UPPER, & LOWER

CHAR_LENGTH('Hello World');	returns number of characters
UPPER('Hello World');	returns 'HELLO WORLD'
LOWER('Hello World');	returns 'hello world'

DISTINCT

SELECT DISTINCT <col name> FROM <table name>	returns values without duplicates
SELECT DISTINCT CONCAT(fname, " ", lname) FROM <table>	returns distinct full names in one column in case two people have the same last name
SELECT DISTINCT fname, lname FROM <table>	- returns distinct full names again - just in two different columns

ORDER BY

SELECT <column> FROM <table> ORDER BY <column>;	returns column from table sorted by... - numbers are smallest to largest - strings sorted A-Z with symbols and number first

SELECT <column> FROM <table> ORDER BY <other col>;	okay to sort by column you're not displaying
SELECT <col> FROM <table> ORDER BY <col> DESC;	reverse order of sort which applies to numbers and strings
SELECT x, y, z FROM table ORDER BY 2;	will display columns x, y, z sorted by y (2nd)
SELECT x, y FROM table ORDER BY y, x;	displays columns x and y ordered by y and any conflicts by x

LIMIT

SELECT title FROM books LIMIT 3;	displays the first 3 titles of books in books table
SELECT title, released_year FROM books ORDER BY released_year DESC LIMIT 5;	displays the title and released year ordered by release year of first 5 books in descending order (newest-oldest)
SELECT title FROM books ORDER BY title LIMIT 1,5;	displays 5 book titles ordered by title and starting at 2nd book

LIKE

SELECT title FROM books WHERE title LIKE '%da%';	- displays all book titles where the title contains the string "da" - can be anywhere because '%' is a wildcard
SELECT stock FROM books WHERE stock LIKE '__';	displays all stock that contains 2 digits. Corresponds to number of '_' and can also work for chars
SELECT title FROM books WHERE title LIKE '%\%%';	showing that you have to use the escape character '\' to actually search for '%'
SELECT title FROM books WHERE title LIKE '%_%'	use escape character to search for '_'

Looking at aggregate functions

COUNT

SELECT COUNT(*) FROM books;	counts number of books
SELECT COUNT(DISTINCT name) FROM books;	counts number of books with distinct name
SELECT COUNT(*) FROM books WHERE title LIKE '%the%';	counts number of books with title containing 'the'

GROUP BY

SELECT title, lname FROM books GROUP BY lname;	displays title and last name of author of book and they are grouped by last name
SELECT lname, COUNT(*) FROM books GROUP BY lname;	displays last name and the count of the number of books in each group; this group is authors w/same last name
SELECT release, COUNT(*) FROM books GROUP BY release;	shows the year and the count of books released in that year

MIN / MAX

SELECT MIN(release);	displays earliest release year
SELECT MAX(release);	displays most recent release year

★ Problem with MIN / MAX to be aware of…

SELECT MAX(pages), title FROM books;	the problem is that we would probably expect this to display the pages and title of the book with the most pages. It does not. It spits on the highest page number and the first title in the database

★ Solutions to this problem…

SELECT * FROM books WHERE pages=(SELECT MIN(pages) FROM books);	this is one solution and MIN can be substituted in no problem. This can be slow b/c it's doing 2 queries
SELECT * FROM books ORDER BY pages ASC LIMIT 1;	better solution and can also easily apply to MIN as well by replacing with DESC

*** Style is going to change because I'm pasting from the course now***

`SELECT author_fname,` `  author_lname,` `  Min(released_year)` `FROM  books` `GROUP  BY author_lname,` `  author_fname;`	displays each author's first book (aka the first released year for their books) groups by author's with the same first and last name (obviously not very likely)
`SELECT author_fname,` ` author_lname,` ` Max(pages)` `FROM books` `GROUP BY author_lname,` `  author_fname;`	displays each author's book with the most pages again grouped by author's with same first and last name
`SELECT COUNT(CustomerID), Country` `FROM Customers` `GROUP BY Country` `ORDER BY COUNT(CustomerID) DESC;`	this is from w3schools.com but is an example of displaying the number of customers in each country ordered largest to smallest

`SELECT` `CONCAT(author_fname, ' ', author_lname) AS author,` `MAX(pages) AS 'longest book'` `FROM books` `GROUP BY author_lname, author_fname;`	just makes it prettier by using CONCAT and AS

SUM

`SELECT SUM(pages)` `FROM books;`	sums up total number of pages in the books database
`SELECT SUM(released_year) FROM books;`	adds up all release years; not useful
`SELECT author_fname, author_lname, Sum(pages)` `FROM books` `GROUP BY author_lname, author_fname;`	sums up all pages written by each author and displays them again use GROUP BY with first and last name to ensure people with same name aren't lumped together

AVG

`SELECT AVG(released_year)` `FROM books;`	displays the average release year of all the books
`SELECT AVG(pages)` `FROM books;`	displays the average number of pages for all the books
`SELECT released_year, AVG(stock_quantity)` `FROM books` `GROUP BY released_year;`	displays the average amount of stock for each release year
`SELECT author_fname, author_lname, AVG(pages) FROM books` `GROUP BY author_lname, author_fname;`	displays the average number of pages for each author and all their books

FINAL THOUGHTS OF AGGREGATE FUNCTIONS….

`SELECT CONCAT(author_fname, ' ', author_lname) FROM books` `ORDER BY pages DESC LIMIT 1;`	displays the author's name with the longest book
`SELECT released_year AS year,` `COUNT(*) AS '# of books',` `AVG(pages) AS 'avg pages'` `FROM books` `GROUP BY released_year;`	displays the released year and then the number of books from that year and the average number of pages of the books in that year all possible because we are grouping by release year

Data Types Continued

CHAR vs VARCHAR

- Both store text
- CHAR has a fixed length
    - e.g. if you have book titles of type CHAR(5), every title will have 5 characters
    - any title entered with more than 5 will be truncated
    - any title entered with less than 5 will have spaces at the end up to 5
- The length of CHAR can be any value from 0 to 255.
- When CHAR values are stored, they are right-padded with spaces to the specified length. When CHAR values are retrieved, trailing spaces are removed unless the PAD_CHAR_TO_FULL_LENGTH SQL mode is enabled.
- Advantages of using CHAR are few but it is faster for fixed length text.
    - e.g. state abbreviations (FL, MO, OH), yes/no flags (Y/N), sex (M/F)

Value	Char(4)[note space]	Storage	Varchar(4)	Storage
‘’	‘ ‘	4 bytes	‘’	1 byte
‘ab’	‘ab ‘	4 bytes	‘ab’	3 bytes
‘abcd’	‘abcd’	4 bytes	‘abcd’	5 bytes
‘abcdefg’	‘abcd’	4 bytes	‘abcdefg’	5 bytes

DECIMAL

- DECIMAL(5, 2)
    - ‘5’ represents the total number of digits before and after decimal point
    - ‘2’ represents digits after decimal
    - so the max decimal number is 999.99
- If you try to add a decimal that exceeds the max number it will just put the max number. In this case that would be 999.99
- If you exceed the number of digits after the decimal it will round
    - e.g. for this case 2.999 will round to 3.00 if you try to insert it

FLOAT vs DOUBLE

- Both store larger numbers using less space than decimal, but they are less precise than decimals
- Numbers may not be as accurate if you are at the cusp

Data Type	Memory Needed	Precision Issues
FLOAT	4 bytes	~7 bytes
DOUBLE	8 bytes	~15 bytes

- Use DECIMALS with large numbers, unless for some reason you don’t think precision is important
- e.g. from course
    - if you enter this number into a column of FLOATs → 8877665544.45
    - when you display it, you can see it becomes 8877670000
    - basically kept the first 6 digits and then rounded

DATE, TIME, and DATETIME

FORMAT	
DATE	'YYYY-MM-DD'
TIME	'HH:MM:SS'
DATETIME	'YYYY-MM-DD HH:MM:SS'

- When entering data into a table be sure to enter it as a string
  - e.g.

```
INSERT INTO people (name, birthdate, birthtime, birthdt)
VALUES('Padma', '1983-11-11', '10:07:35', '1983-11-11
10:07:35');
```

CURDATE, CURTIME, and NOW

CURDATE	gives current date
CURTIME	gives current time
NOW	gives current datetime

FORMATTING DATES

- There are a ton of date and time functions out there to use that help us gain even more data from date and time related objects we mentioned right above.

`SELECT name, birthdate, DAY(birthdate) FROM people;`	displays 3 columns with the name, birthday, and the day of month (using DAY) of the birthdate
`SELECT name, birthdate, DAYNAME(birthdate) FROM people;`	displays the same thing, except the DAYNAME displays the day of week it is (e.g. Monday, Friday)
`SELECT name, birthdate, DAYOFWEEK(birthdate) FROM people;`	DAYOFWEEK will give you the day number in a week where Sunday is 1 and Saturday is 7
`SELECT name, birthdate, DAYOFYEAR(birthdate) FROM people;`	DAYOFYEAR displays what number day in a year e.g. 1943-12-25 would be the 359th day of the year
`SELECT name, birthtime, DAYOFYEAR(birthtime) FROM people;`	will display NULL because there is no day in the time object
`SELECT name, birthdt, MONTH(birthdt) FROM people;`	MONTH shows the number of month it is
`SELECT name, birthdt,`	MONTHNAME shows the month name

`MONTHNAME(birthdt) FROM people;`	
`SELECT name, birthtime, HOUR(birthtime) FROM people;`	HOUR displays just the hour in the time
`SELECT name, birthtime, MINUTE(birthtime) FROM people;`	same but for minute instead
`SELECT DATE_FORMAT(birthdt, 'Was born on a %W') FROM people;`	DATE_FORMAT allows us to format which data is displayed. The '%W' means the name of the day of week
`SELECT DATE_FORMAT(birthdt, '%m/%d/%Y') FROM people;`	displays normal date like 02/23/1996
`SELECT DATE_FORMAT(birthdt, '%m/%d/%Y at %h:%i') FROM people;`	displays the same date as above along with the time formatted

DATE MATH

`SELECT DATEDIFF(NOW(), birthdate) FROM people;`	returns number of days between now and the birth date
`SELECT birthdt, DATE_ADD(birthdt, INTERVAL 1 MONTH) FROM people;`	DATE_ADD is for adding units of time to the specified datetime. This case we are adding one month to the dates and displaying it.
`SELECT birthdt, DATE_ADD(birthdt, INTERVAL 10 SECOND) FROM people;`	e.g. with SECOND instead
`SELECT birthdt, DATE_ADD(birthdt, INTERVAL 3 QUARTER) FROM people;`	e.g. with 9 months (aka 3 QUARTER)
`SELECT birthdt, birthdt + INTERVAL 1 MONTH FROM people;`	instead of using DATE_ADD you can use '+' and instead of DATE_SUB you can use '-'
`SELECT birthdt, birthdt - INTERVAL 5 MONTH FROM people;`	e.g. with subtracting
`SELECT birthdt, birthdt + INTERVAL 15 MONTH + INTERVAL 10 HOUR FROM people;`	e.g. adding multiple units MONTH and HOUR

TIMESTAMP

- The TIMESTAMP data type contains date and time (like DATETIME) and has a range between 1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC
    - basically when it runs out of bytes

`CREATE TABLE comments ( content VARCHAR(100), created_at TIMESTAMP DEFAULT NOW() );`	creates at column called "created_at" of type TIMESTAMP and the DEFAULT action when a new entry is added is to take the datetime data using NOW( )

`CREATE TABLE comments2 (` `  content VARCHAR(100),` `  changed_at TIMESTAMP DEFAULT NOW() ON UPDATE CURRENT_TIMESTAMP` `);`	does the same thing, except the 'ON UPDATE CURRENT_TIMESTAMP' will update the timestamp if the row is changed CURRENT_TIMESTAMP and NOW() would be the same in this case

Logical Operators

= and !=

`SELECT title FROM books WHERE released_year = 2017;`	pretty simple, displays books released in 2017 NOTE: for equal it is only a single equal sign not two
`SELECT title FROM books WHERE released_year != 2017;`	displays books not released in 2017
`SELECT title, author_lname FROM books WHERE author_lname != 'Harris';`	displays books where the author's last name is not 'Harris'

LIKE and NOT LIKE

`SELECT title FROM books WHERE title LIKE 'W%';`	displays books starting with a 'W' (not case sensitive)
`SELECT title FROM books WHERE title LIKE '%W%';`	displays books that contain a 'W'
`SELECT title FROM books WHERE title NOT LIKE 'W%';`	displays books that don't contain a 'W'

> and >= and < and <=

`SELECT title, released_year FROM books` `WHERE released_year > 2000 ORDER BY released_year;`	displays all books released after 2000 and then orders it from least to greatest
`SELECT title, released_year FROM books` `WHERE released_year >= 2000 ORDER BY released_year;`	displays all books released after, and including, 2000 and orders
`SELECT title, stock_quantity FROM books WHERE stock_quantity >= 100;`	displays books with stock greater than or equal to 100
`SELECT 99 > 567;`	returns 0 (false)
`SELECT 'a' > 'b';`	returns 0
`SELECT 'A' >= 'a';`	returns 1 because 'A' is equal to 'a' in MySQL
`SELECT 'h' < 'p';`	returns 1 (true)

&& or AND

- Note that '&&' is deprecated in newer versions of MySQL

`SELECT` `  title,` `  author_lname,` `  released_year FROM books` `WHERE author_lname='Eggers'` `  AND released_year > 2010;`	returns the books written by author with last name 'Eggers' AND released after 2010
`SELECT 1 < 5 && 7 = 9;`	false
`SELECT *` `FROM books` `WHERE author_lname='Eggers'` `  AND released_year > 2010` `  AND title LIKE '%novel%';`	returns books written by author with last name 'Eggers' AND released year is after 2010 AND the title contains word 'novel'

|| or OR

- Note that '||' is deprecated in newer versions of MySQL

`SELECT` `  title,` `  author_lname,` `  released_year` `FROM books` `WHERE author_lname='Eggers' ||` `released_year > 2010;`	all books written by 'Eggers' or any book written after 2010
`SELECT 40 <= 100 || -2 > 0;`	returns 1 (true)
`SELECT title,` `  author_lname,` `  released_year,` `  stock_quantity` `FROM  books` `WHERE  author_lname = 'Eggers'` `  || released_year >` `2010` `OR  stock_quantity > 100;`	returns books that are either by 'Eggers', released after 2010, or have a stock greater than 100

BETWEEN

`SELECT title, released_year FROM` `books WHERE released_year >= 2004` `&& released_year <= 2015;`	how to accomplish what BETWEEN does
`SELECT title, released_year FROM` `books` `WHERE released_year BETWEEN 2004` `AND 2015;`	does the same thing as the above example, just using the BETWEEN operator

- When using BETWEEN with date or time values, use CAST() to explicitly convert the values to the desired data type

`SELECT CAST('2017-05-02' AS DATETIME);`	converts a date to a datetime type result will be '2017-05-02 00:00:00'
`SELECT name, birthdt FROM people WHERE birthdt BETWEEN '1980-01-01' AND '2000-01-01';`	displays people with birthday between 1980 and 2000 **still works, but we should cast to datetime to avoid problems**
`SELECT name, birthdt FROM people WHERE birthdt BETWEEN CAST('1980-01-01' AS DATETIME) AND CAST('2000-01-01' AS DATETIME);`	shows how to properly do the same thing as above more fail-proof

IN and NOT IN

`SELECT title, author_lname FROM books WHERE author_lname='Carver' OR author_lname='Lahiri' OR author_lname='Smith';`	showing that we can do it without the IN operator
`SELECT title, author_lname FROM books WHERE author_lname IN ('Carver', 'Lahiri', 'Smith');`	simplified version of above that will display authors with last name 'Carver', 'Lahiri', and 'Smith'
`SELECT title, released_year FROM books WHERE released_year IN (2017, 1985);`	shows books released in years 2017 and 1985
`SELECT title, released_year FROM books WHERE released_year NOT IN (2000,2002,2004,2006,2008,2010,2012,2014,2016);`	displays books released not in even number years between 2000 and 2016
`SELECT title, released_year FROM books WHERE released_year >= 2000 AND released_year % 2 != 0;`	much simpler way to take advantage of modulo e.g. 2000 / 2 gives remainder of 0 so this is false e.g. 2001 / 2 gives remainder of 1 so this is true
`SELECT title, released_year FROM books WHERE released_year >= 2000 AND released_year % 2 != 0 ORDER BY released_year;`	same as above and it orders by released year least to greatest
`SELECT title, author_lname FROM books WHERE author_lname='Carver' OR author_lname='Lahiri' OR author_lname='Smith';`	showing that we can do it without the IN operator

`SELECT title, author_lname FROM books WHERE author_lname IN ('Carver', 'Lahiri', 'Smith');`	simplified version of above that will display authors with last name 'Carver', 'Lahiri', and 'Smith'
`SELECT title, released_year FROM books WHERE released_year IN (2017, 1985);`	shows books released in years 2017 and 1985
`SELECT title, released_year FROM books WHERE released_year NOT IN (2000,2002,2004,2006,2008,2010,2012,2014,2016);`	displays books released not in even number years between 2000 and 2016
`SELECT title, released_year FROM books WHERE released_year >= 2000 AND released_year % 2 != 0;`	much simpler way to take advantage of modulo e.g. 2000 / 2 gives remainder of 0 so this is false e.g. 2001 / 2 gives remainder of 1 so this is true
`SELECT title, released_year FROM books WHERE released_year >= 2000 AND released_year % 2 != 0 ORDER BY released_year;`	same as above and it orders by released year least to greatest

CASE statements

`SELECT title, released_year,` `  CASE` `  WHEN released_year >= 2000 THEN 'Modern Lit'` `  ELSE '20th Century Lit'` `  END AS GENRE` `FROM books;`	the CASE creates a third column and says that any book released 2000 on is considered 'Modern Lit' and anything else, which is before 2000, would be considered '20th Century Lit' don't forget END
`SELECT title, stock_quantity,` `  CASE` `  WHEN stock_quantity BETWEEN 0 AND 50 THEN '*'` `  WHEN stock_quantity BETWEEN 51 AND 100 THEN '**'` `  ELSE '***'` `  END AS STOCK` `FROM books;`	pretty cool way of visualizing the stock anything between 0-50 gets a '*' anything between 50-100 gets a '**' anything else gets a '***'
`SELECT title, stock_quantity,` `  CASE` `  WHEN stock_quantity <= 50 THEN '*'` `  WHEN stock_quantity <= 100 THEN '**'` `  ELSE '***'` `  END AS STOCK` `FROM books;`	just showing another way of doing it without using BETWEEN this works because we know that if we make it to the second statement it will definitely be something greater than 50 because it will break out if it's under 50

Interesting Example from the Lesson's Final Challenge

`SELECT author_fname, author_lname,` `  CASE` `  WHEN COUNT(*) = 1 THEN '1 book'` `  ELSE CONCAT(COUNT(*), ' books')` `  END AS COUNT` `FROM books` `GROUP BY author_lname, author_fname;`	displays the number of books each author has written

Starting to look at real data!

- Seeing how tables can be connected together
- Real data is messy!
- Relationships and Joins

Data Relationships

1. One to One Relationship
    a. e.g. one customer has one row with customer data and one row of customer data has one customer associated
2. One to Many Relationship
    a. most common
    b. e.g. a book has thousands of reviews
    c. one book has many reviews but the reviews belong to one book
3. Many to Many Relationship
    a. fairly common
    b. book has multiple authors and authors can have multiple books

One to Many Relationship Example

- Customers and Orders (2 tables) / 1 customer can have many orders / every order belongs to 1 person
- We Want to Store…
    - A customer's first and last name
    - A customer's email
    - The date of the purchase
    - The price of the order
- So we could store this in one table, but there would be problems….
    - duplicate data if there was multiple orders
    - a customer hasn't made an order yet so date and price are NULL
- Better to keep customers and orders separate!
- Should have two tables

Customers	Orders
customer_id	order_id
first_name	order_date
last_name	amount
email	customer_id

- Notice the reference (customer_id) in both tables to have that relationship
- PRIMARY KEY is important because it keeps data separate and unique
    - created when the entry is made by auto-incrementing
- FOREIGN KEY is the customer_id in the Orders table
    - references to another table within a given table
    - e.g. of working with a foreign key

`CREATE TABLE customers(` `  id INT AUTO_INCREMENT` `PRIMARY KEY,` `  first_name VARCHAR(100),` `  last_name VARCHAR(100),` `  email VARCHAR(100)` `);`	creates the customer table with an auto incrementing primary key called 'id'
`CREATE TABLE orders(` `  id INT AUTO_INCREMENT` `PRIMARY KEY,` `  order_date DATE,` `  amount DECIMAL(8,2),` `  customer_id INT,` `  FOREIGN KEY(customer_id)` `REFERENCES customers(id)` `);`	creates the id creates FOREIGN KEY where the name in the parentheses is the name of the foreign key in this table REFERENCES is what points at the other table and the name of the column we're looking for in that table ('id' in this case)
`INSERT INTO customers` `(first_name, last_name,` `email)` `VALUES ('Boy', 'George',` `'george@gmail.com'),` `  ('George', 'Michael',` `'gm@gmail.com'),` `  ('David', 'Bowie',` `'david@gmail.com'),` `  ('Blue', 'Steele',` `'blue@gmail.com'),` `  ('Bette', 'Davis',` `'bette@aol.com');`	inserts some customers note: the id is auto-incrementing and something we don't have to explicitly do
`INSERT INTO orders` `(order_date, amount,` `customer_id)` `VALUES` `('2016/02/10', 99.99, 1),` `('2017/11/11', 35.50, 1),` `('2014/12/12', 800.67, 2),` `('2015/01/03', 12.50, 2),` `('1999/04/11', 450.25, 5);`	inserts some orders the last value for each order is associated to a value in the customers table
`INSERT INTO orders` `(order_date, amount,` `customer_id)` `VALUES` `('2016/06/06', 33.67, 98);`	this will fail because there is currently not 98 customers we only have 5 right now

Example of using data from both tables

- SELECT id FROM customers WHERE last_name='George'; - SELECT * FROM orders WHERE customer_id = 1;	finding orders placed by George in a 2-step process
SELECT * FROM orders WHERE customer_id = (SELECT id FROM customers WHERE last_name='George'););	using a subquery we can do it in one eloquent statement basically only displays the orders placed with a customer id retrieved from the customer table

Cross Join

SELECT * FROM customers, orders;	the **CROSS JOIN** is used to generate a paired combination of each row of the first table with each row of the second table basically useless

Implicit Inner Join

SELECT * FROM customers, orders WHERE customers.id = orders.customer_id;	similar to the Cross Join in the sense that it displays all data, but the only difference is we are only displaying the entries where the id in the customers table (hence customers.id) is equal to the customer_id in the orders table (hence orders.cutomer_id)
SELECT first_name, last_name, order_date, amount FROM customers, orders WHERE customers.id = orders.customer_id;	basically the same as above except we are getting rid of all the id columns

Explicit Inner Join

- explicitly says JOIN

SELECT first_name, last_name, order_date, amount FROM customers JOIN orders ON customers.id = orders.customer_id;	gives the same table as above but explicitly says JOIN

Getting fancy

SELECT first_name, last_name, SUM(amount) AS total_spent FROM customers JOIN orders ON customers.id = orders.customer_id GROUP BY orders.customer_id ORDER BY total_spent DESC;	displays the total amount spent by customers and ordered greatest to least GROUP BY orders.customer_id because we know these are unique and first/last names could be duplicated in the data set

Left Join

- takes everything from one table, as well as any matching records in the second table
- in this example we are displaying all customers, whether they have placed an order or not. Their order data would just be NULL

`SELECT first_name, last_name, order_date, amount FROM customers LEFT JOIN orders ON customers.id = orders.customer_id;`	displays all customers whether they ordered anything or not if they haven't ordered anything it is displayed as NULL notice the LEFT JOIN
`SELECT first_name, last_name, IFNULL(SUM(amount), 0) AS total_spent FROM customers LEFT JOIN orders ON customers.id = orders.customer_id GROUP BY customers.id ORDER BY total_spent;`	displays first name, last name, and the sum of total amount spent by each customer IFNULL - if sum amount is NULL then we display it as 0 instead again, GROUP BY customers.id (pretty sure orders.customer_id would work as well) because that's better than first name/ last name

Right Join

- goes from the other table (orders in this case)
- shouldn't have a difference in this case because we don't have any orders without a customer
- if you try to delete a customer it will throw an error because of the foreign key in orders
- in this example you'd have to get rid of the foreign key to make a RIGHT JOIN work

On Delete Cascade

- when we have a foreign key, if we want to delete a customer from the table it won't let us because of the foreign key in the orders table
- when you specify a foreign key you have to put ON DELETE CASCADE

`CREATE TABLE customers( id INT AUTO_INCREMENT PRIMARY KEY, first_name VARCHAR(100), last_name VARCHAR(100), email VARCHAR(100) );` `CREATE TABLE orders( id INT AUTO_INCREMENT PRIMARY KEY, order_date DATE, amount DECIMAL(8,2), customer_id INT, FOREIGN KEY(customer_id) REFERENCES customers(id) ON DELETE CASCADE );`	note in the orders table we added the ON DELETE CASCADE this means if we delete a customer the corresponding orders in the order table will be deleted as well

One to Many Final Exercise Interesting Solutions

`SELECT first_name,` `  Ifnull(Avg(grade), 0) AS average,` `  CASE` `  WHEN Avg(grade) IS NULL THEN 'FAILING'` `  WHEN Avg(grade) >= 75 THEN 'PASSING'` `  ELSE 'FAILING'` `  end AS passing_status` `FROM students` `  LEFT JOIN papers` `  ON students.id = papers.student_id` `GROUP BY students.id` `ORDER BY average DESC;`	displays a table with 3 columns 1) as name 2) prints the average paper grade for each student 3) prints whether the student is passing or failing based off the average paper grade

Many to Many Relationship Example

- some examples
  - Books <-> Authors - books can have multiple authors and authors can have multiple books
  - Blog posts <-> Tags - posts have multiple tags and tags have multiple posts
  - Students <-> Classes - students have multiple classes and classes have multiple people
- example for this section
  - imagine a tv show reviewing application
  - reviewers, tv shows, and reviews
  - need 3 tables: tv series data ← review data → reviewer data
  - so the way the tv series data is connected to the reviewer data is through the reviews
  - the schema looks like this

Reviewers	Reviews	Series
id	id	id
first_name	rating	title
last_name	series_id	released_year
	reviewer_id	genre

  - the two foreign keys -- series_id and reviewer_id -- are what connects the tables

Challenge 1

`SELECT` `  title,` `  rating` `FROM series` `JOIN reviews` `  ON series.id = reviews.series_id;`	displays the series that have reviews and each review for each series

Challenge 2

```
SELECT
   title,
   AVG(rating)AS avg_rating
FROM series
JOIN reviews
   ON series.id =
reviews.series_id
GROUP BY series.id
ORDER BY avg_rating;
```

displays the series title along with the average rating of the show

ordered in ascending order

GROUP BY series.id in case the series title has repeats

Challenge 3

```
SELECT title AS unreviewed_series
FROM series
LEFT JOIN reviews
   ON series.id =
reviews.series_id
WHERE rating IS NULL;
```

displays the series that haven't been reviewed yet

do a LEFT JOIN because we care about all the data in the first table (series) and want to look at the overlap, or lack of overlap, in the reviews table

Challenge 4

```
SELECT genre,
      Round(Avg(rating), 2) AS
avg_rating
FROM   series
      INNER JOIN reviews
              ON series.id =
reviews.series_id
GROUP  BY genre;
```

displays the average rating for each genre

Challenge 5

```
SELECT first_name,
      last_name,
      Count(rating)
AS COUNT,
      Ifnull(Min(rating), 0)
AS MIN,
      Ifnull(Max(rating), 0)
AS MAX,
      Round(Ifnull(Avg(rating),
0), 2)            AS AVG,
      IF(Count(rating) > 0,
'ACTIVE', 'INACTIVE') AS STATUS
FROM   reviewers
      LEFT JOIN reviews
             ON reviewers.id =
reviews.reviewer_id
GROUP  BY reviewers.id;
```

displays a lot of data

- reviewer first name
- reviewer last name
- count of how many reviews the reviewer has made
- their minimum rating
- their maximum rating
- their average rating
- if they are an active user or not
    - had at least 1 review

SELECT first_name, last_name, Count(rating) AS COUNT, Ifnull(Min(rating), 0) AS MIN, Ifnull(Max(rating), 0) AS MAX, Round(Ifnull(Avg(rating), 0), 2) AS AVG, CASE WHEN Count(rating) >= 10 THEN 'POWER USER' WHEN Count(rating) > 0 THEN 'ACTIVE' ELSE 'INACTIVE' end AS STATUS FROM reviewers LEFT JOIN reviews ON reviewers.id = reviews.reviewer_id GROUP BY reviewers.id;	does the same as the above except for the CASE it will say 'POWER USER' if a reviewer has more than 10 reviews, 'ACTIVE' between 1-9 reviews, 'INACTIVE' for 0 reviews

Challenge 6

SELECT title, rating, CONCAT(first_name,' ', last_name) AS reviewer FROM reviewers INNER JOIN reviews ON reviewers.id = reviews.reviewer_id INNER JOIN series ON series.id = reviews.series_id ORDER BY title;	Joining 3 tables! To do that you have to join the 2 foreign keys in the 'reviews' table to the other two tables - 'reviewers' and 'series' Pretty nifty stuff.

Looking at Instagram Data

- Information we will need for the schema: users, photos, comments, likes, hashtags, and followers/following
- Creating tables
    - Users

id
username
created_at

```
CREATE TABLE users (
   id INTEGER AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(255) UNIQUE NOT NULL,
   created_at TIMESTAMP DEFAULT NOW()
);
```

    - Photos

id
image_url
user_id
created_at

```
CREATE TABLE photos (
   id INTEGER AUTO_INCREMENT PRIMARY KEY,
   image_url VARCHAR(255) NOT NULL,
   user_id INTEGER NOT NULL,
   created_at TIMESTAMP DEFAULT NOW(),
   FOREIGN KEY(user_id) REFERENCES
users(id)
);
```

    - Comments

id
comment_text
user_id
photo_id
created_at

```
CREATE TABLE comments (
   id INTEGER AUTO_INCREMENT PRIMARY KEY,
   comment_text VARCHAR(255) NOT NULL,
   photo_id INTEGER NOT NULL,
   user_id INTEGER NOT NULL,
   created_at TIMESTAMP DEFAULT NOW(),
   FOREIGN KEY(photo_id) REFERENCES
photos(id),
   FOREIGN KEY(user_id) REFERENCES
users(id)
);
```

    - Likes

user_id
photo_id
created_at

```
CREATE TABLE likes (
   user_id INTEGER NOT NULL,
   photo_id INTEGER NOT NULL,
   created_at TIMESTAMP DEFAULT NOW(),
   FOREIGN KEY(user_id) REFERENCES
users(id),
   FOREIGN KEY(photo_id) REFERENCES
photos(id),
   PRIMARY KEY(user_id, photo_id)
);
```

        ★ the important thing to note here is the last line with the PRIMARY KEY. Since we don't want a user to be able to like a photo over and over (aka one like per user per photo), this PRIMARY KEY line accomplishes that.

- Followers/Following

follower_id followee_id created_at	CREATE TABLE follows (follower_id INTEGER NOT NULL, followee_id INTEGER NOT NULL, created_at TIMESTAMP DEFAULT NOW(), FOREIGN KEY(follower_id) REFERENCES users(id), FOREIGN KEY(followee_id) REFERENCES users(id), PRIMARY KEY(follower_id, followee_id));

- Hashtags

id tag_name created_at	CREATE TABLE tags (id INTEGER AUTO_INCREMENT PRIMARY KEY, tag_name VARCHAR(255) UNIQUE, created_at TIMESTAMP DEFAULT NOW());

- Photo_Hashtags

photo_id tag_id	CREATE TABLE photo_tags (photo_id INTEGER NOT NULL, tag_id INTEGER NOT NULL, FOREIGN KEY(photo_id) REFERENCES photos(id), FOREIGN KEY(tag_id) REFERENCES tags(id), PRIMARY KEY(photo_id, tag_id));

<u>Looking at different problems within IG data</u>

Challenge 1

- Find the 5 longest users.

SELECT * FROM users ORDER BY created_at LIMIT 5;	easy one

Challenge 2

- Need to figure out the best time to schedule an ad campaign. Which day of the week do most users register on?

SELECT DAYNAME(created_at) AS day, COUNT(*) AS total FROM users GROUP BY day ORDER BY total DESC LIMIT 2;	since we're using GROUP BY we can do the COUNT(*) without it counting all the users like I originally though

Challenge 3

- We want to target inactive users. Find the users who have never posted a photo before.

`SELECT username` `FROM users` `LEFT JOIN photos` `  ON users.id = photos.user_id` `WHERE photos.id IS NULL;`	pretty easy one using the WHERE is the key part here

Challenge 4

- Running a contest to see who can get the most likes on their photo. Who won?

`SELECT` `  photos.id,` `  photos.image_url,` `  likes.user_id` `FROM photos` `INNER JOIN likes` `  ON likes.photo_id = photos.id;`	this one is tricky so I'm breaking it up we are displaying the photo id, photo url, and the user id of the user who likes the photo generates a big table of all photos and all their likes
`SELECT` `  photos.id,` `  photos.image_url,` `  COUNT(*) AS total` `FROM photos` `INNER JOIN likes` `  ON likes.photo_id = photos.id` `GROUP BY photos.id` `ORDER BY total DESC` `LIMIT 1;`	next step we need to limit the data down and find the photo with most likes GROUP BY the photos.id and then COUNT that number of times the photos.id appears ORDER BY greatest to least and limit to 1 Now we just need to join the user data to convert the photo.id to a username
`SELECT` `  username,` `  photos.id,` `  photos.image_url,` `  COUNT(*) AS total` `FROM photos` `INNER JOIN likes` `  ON likes.photo_id = photos.id` `INNER JOIN users` `  ON photos.user_id = users.id` `GROUP BY photos.id` `ORDER BY total DESC` `LIMIT 1;`	by joining the users table in their ew can access the username and BOOM it's done

Challenge 5

- How many times does the average user post?

`SELECT` `(SELECT Count(*) FROM  photos) /` `(SELECT Count(*) FROM  users)` `AS avg;`	use subqueries

Challenge 6

- What are the top 5 most commonly used hashtags?

`SELECT tags.tag_name,` `  **Count**(*) AS total` `FROM  photo_tags` `JOIN tags ON photo_tags.tag_id = tags.id` `GROUP  BY tags.id` `ORDER  BY total DESC` `LIMIT  5;`	straight forward gathering that we would rather GROUP BY id rather than name

Challenge 7

- Find users who have liked every single photo.

`SELECT` `  username,` `  Count(*) AS num_likes` `FROM  users` `INNER JOIN likes ON users.id = likes.user_id` `GROUP  BY likes.user_id` `HAVING num_likes = (SELECT Count(*)FROM  photos);`	without the HAVING clause at the end this would display all users and the total number of likes they have the HAVING function was added to allow for aggregate functions in this HAVING function we are only displaying the users that have their 'num_likes' equal to the total number of photographs (257) in the photos table

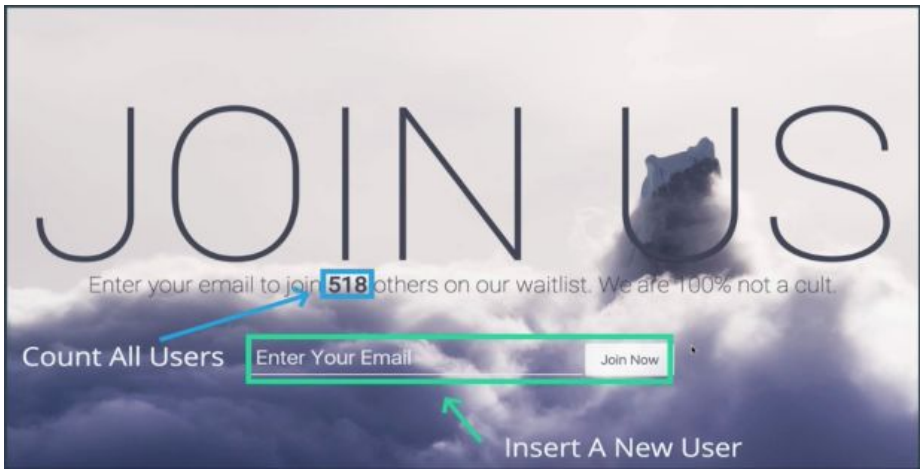
★ VERY USEFUL THING FOUND RIGHT ABOVE → HAVING

<u>Starting to learn how to build a web application</u>

Connecting MySQL to another language (Node.js)

- e.g. Client Computer <--> Node.js <--> MySQL Database
- basically a language like Node.js is what talks to the MySQL database
- doesn't have to be Node.js. Could be a C language, python, Ruby, PHP, Java, etc

Join Us App

- A startup mailing list application
- Point is to gather people's email addresses
- 

- Workflow of this web application
    - Client computer <--> Node.js <--> MySQL Database
    - number of users would just be a count in the ‘users’ table
    - inserting a new user into the ‘user’ table would have to account for repeats

Going to start with the relationship between Node.js <--> MySQL Database

- Difference between Javascript and Node.js
    - Node is Javascript but you can use it on the backend
    - Traditionally Javascript was just used on the client side to make pages pretty and functional
    - Node.js was made to help interact on the backend

`for(var i = 0; i < 500; i++){` `  console.log("HELLO WORLD!");` `}`	**//Print "HELLO WORLD" 500 times using Node**
`node filename.js`	**// Execute file with**

- FAKER - a tool to generate fake data for our web application

`npm install faker`	**// Install Faker via command line**
`var faker = require('faker');`	**// Require it inside of a JS file**
`console.log(faker.internet.email());`	**// Print a random email**
`console.log(faker.date.past());`	**// Print a random past date**
`console.log(faker.address.city());`	**// Print a random city**
`function generateAddress()` `{` `console.log(faker.address.streetAddress()` `);` `console.log(faker.address.city());` `console.log(faker.address.state());` `}`	**// We can define a new function**

- Introducing MySQL to Node.js

`npm install mysql`	Install the MySQL Node Package
`var mysql = require('mysql');` `var connection = mysql.createConnection({` `  host     : 'localhost',` `  user     : 'root',     // your root username` `  database : 'join_us'   // the name of your db` `});`	Connect to database Will need to add a 'password' field after username
`connection.query('SELECT 1 + 1 AS solution', function (error, results, fields) {` `  if (error) throw error;` `  console.log('The solution is: ', results[0].solution);` `});`	Running a simple query Notice how we're using an alias in our query and then accessing it in results later on
`var q = 'SELECT CURTIME() as time, CURDATE() as date, NOW() as now';` `connection.query(q, function (error, results, fields) {` `  if (error) throw error;` `  console.log(results[0].time);` `  console.log(results[0].date);` `  console.log(results[0].now);` `});`	Another query, selecting 3 things

Schema for our Join Us Project

`CREATE TABLE users (` `  email VARCHAR(255) PRIMARY KEY,` `  created_at TIMESTAMP DEFAULT NOW()` `);`	pretty simple
`var q = 'SELECT COUNT(*) AS total FROM users ';` `connection.query(q, function (error, results, fields) {` `  if (error) throw error;` `  console.log(results[0].total);` `});`	what our query will look like in Node.js to our MySQL database to get the number of users

Inserting Users

`` var person = { email: faker.internet.email(), created_at: faker.date.past() }; var end_result = connection.query('INSERT INTO users SET ?', person, function(err, result) { if (err) throw err; console.log(result); }); ```	using the 'var person' we can set the two columns in the table we're inserting into 'email' and 'created_at' to the faker random values then for the query do a 'INSERT INTO users SET ?' to pass those values in manually passing in the faker.date.past( ) without using the 'var person' won't work. Have to do it this way so the library can work its magic
`` var mysql = require('mysql'); var faker = require('faker'); var connection = mysql.createConnection({ host : 'localhost', user : 'root', database : 'join_us' }); var data = []; for(var i = 0; i < 500; i++){ data.push([faker.internet.email(), faker.date.past()]); } var q = 'INSERT INTO users (email, created_at) VALUES ?'; connection.query(q, [data], function(err, result) { console.log(err); console.log(result); }); connection.end(); ```	final code for inserting tons of users

Now we have all the data, time for some challenges with it

- find the earliest date a user joined

`` SELECT DATE_FORMAT(MIN(created_at), "%M %D %Y") as earliest_date FROM users; ```	pretty simple

- find the email of the first user

`SELECT * FROM users WHERE created_at = (SELECT Min(created_at) FROM users);`	this is how he did it… not how I did and I think my way is easier since it doesn't use subqueries like this
`SELECT * FROM users ORDER BY created_at LIMIT 1;`	my way which seems better

- count the number of users joined in each month

`SELECT Monthname(created_at) AS month, Count(*) AS count FROM users GROUP BY month ORDER BY count DESC;`	same as how I did it

- count number of users with yahoo emails

`SELECT Count(*) AS yahoo_users FROM users WHERE email LIKE '%@yahoo.com';`	also how I did it

- calculate the total number of users per email provider

`SELECT CASE WHEN email LIKE '%@gmail.com' THEN 'gmail' WHEN email LIKE '%@yahoo.com' THEN 'yahoo' WHEN email LIKE '%@hotmail.com' THEN 'hotmail' ELSE 'other' end AS provider, Count(*) AS total_users FROM users GROUP BY provider ORDER BY total_users DESC;`	same as I did as well overall, these were too easy

WEB APP TIME

Express

- a framework to help build web applications
- a backend web application framework for Node.js

NPM Init and package.json files

- create a new folder for the JoinUs web app
- run npm init to create a package.json file
    - helps keep a record of what packages have been installed
    - if the code is sent to someone else all they would need to do is run "npm install' to automatically go and grab all the packages required for the project
    - when doing 'npm init' he changes entry point from 'index.js' to 'app.js'. Don't think it's a big deal

- to save a record of the packages you install with npm, run 'npm install <package name> --save' and it will add the package and version to the package.json file

First Simple Web App

`var express = require('express');` `var app = express();` `app.get("/", function(req, res){` `res.send("HELLO FROM OUR WEB APP!");` `});` `app.listen(8080, function () {` `console.log('App listening on port 8080!');` `});`	the app.get is saying when we get a request for the '/' (aka the homepage) we do a 'res.send' and in this case we are sending a 'HELLO' statement the app is listening on port 8080 so simply go to a web browser and type in 'localhost:8080' to get the page

Adding another route

`app.get("/joke", function(req, res){` `var joke = "What do you call a dog that does magic tricks? A labracadabrador.";` `res.send(joke);` `});`	just adding another route or path simply add a '/joke' to the end of the path
`app.get("/random_num", function(req, res){` `var num = Math.floor((Math.random() * 10) + 1);` `res.send("Your lucky number is " + num);` `});`	add '/random_num' to the end of the path apparently Math.random() returns a number between 0-1 so we have to multiply by 10, round down and add 1 to get the range we want 1-10

Connecting Express and MySQL to get number of users

`app.get("/", function(req, res){` `var q = 'SELECT COUNT(*) as count FROM users';` `connection.query(q, function (error, results) {` `if (error) throw error;` `var msg = "We have " results[0].count + " users";` `res.send(msg);` `});` `});`	puts the MySQL query into the app.get part and the res.send inside the connection.query

- Now that we have the site sort of functional with the grabbing of the total number of users, it's time to start making it look sort of pretty…

- EJS - Embedded Javascript - which allows us to embed javascript into html

```
<h1>JOIN US</h1>

<p class="lead">Enter your email to join
<strong><%= data %></strong>
   others on our waitlist. We are 100% not a
cult. </p>

<form method="POST" action='/register'>
   <input type="text" class="form" name="email"
placeholder="Enter Your Email">
   <button>Join Now</button>
</form>
```

this is the EJS file where we write our html

in here we have the '<%= count %>' which is the count variable we have send over from our app.js file that can be seen below

```
app.set('view engine', 'ejs');
app.get("/", function (req, res) {
   var q = 'SELECT COUNT(*) AS count FROM
users';
   connection.query(q, function (error, result)
{
       if (error) throw error
       var count = result[0].count;
       res.render('home', { data: count });
   })
});
```

couple things to note here

1. the 'app.set' tells the EJS where to find the .ejs file. It will look in a folder called 'views' for a file specified in the 'res.render'
2. in our call for 'res.render' we specify 'home' so the app.js will look for a file in views/home.ejs

the views/home.ejs is show above

we pass the var 'data' from the .js file here to the .ejs file above which allows us to display the count of the number of users

Now we are connecting the form to the MySQL database

```
app.post('/register', function (req, res) {
   var person = { email: req.body.email };
   connection.query('INSERT INTO users SET ?',
person, function (err, result) {
       console.log(err);
       console.log(result);
       res.redirect("/");
   });
});
```

when the submit is hit a POST request is sent with our email to add to the database

we had to add a separate library called body-parser to do the 'var person' line which is completely needed to get that email out of the post request

res.redirect line will help us refresh with our updated user amount

```
<form method="POST" action='/register'>
   <input type="text" class="form" name="email"
placeholder="Enter Your Email">
   <button>Join Now</button>
</form>
```

this is the POST request in the .ejs file

the important thing to note here is the 'name="email"' because that's what we use in the .js file on the 'var person' line above

Database Triggers

- SQL statements that are automatically ran when a specific table is changed
- The syntax...

```
CREATE TRIGGER trigger_name
   trigger_time trigger_event ON table_name FOR EACH ROW
   BEGIN
   …….
   END;
```

    - trigger_time → BEFORE or AFTER
    - trigger_event → INSERT or UPDATE or DELETE
    - table_name → photos or users
- Uses of triggers…
    - validating data
    - manipulating other tables based off an even in another table
- e.g. of a trigger

```
DELIMITER $$

CREATE TRIGGER must_be_adult
     BEFORE INSERT ON users FOR EACH ROW
     BEGIN
          IF NEW.age < 18
          THEN
               SIGNAL SQLSTATE '45000'
                    SET MESSAGE_TEXT =
'Must be an adult!';
          END IF;
     END;
$$

DELIMITER ;
```

DELIMITER changes the delimiter to '$$' so everything is ran at once until that '$$' shows up at the bottom

CREATE TRIGGER creates the trigger which in this case is BEFORE any INSERT on USERS and the FOR EACH ROW is needed just cause

we are checking if the NEW data 'NEW.age' is under 18 and if it is SIGNAL an error ('45000')

change the delimiter back to normal

- e.g. of another trigger to prevent self follows in our instagram table

```
DELIMITER $$

CREATE TRIGGER example_cannot_follow_self
     BEFORE INSERT ON follows FOR EACH ROW
     BEGIN
          IF NEW.follower_id = NEW.following_id
          THEN
               SIGNAL SQLSTATE '45000'
                    SET MESSAGE_TEXT = 'Cannot
follow yourself, silly';
          END IF;
     END;
$$

DELIMITER ;
```

pretty simple trigger

if the follower id and the followee id are the same we shouldn't' allow that to happen

- logging unfollows

`DELIMITER $$` `CREATE TRIGGER create_unfollow` `  AFTER DELETE ON follows FOR EACH ROW` `BEGIN` `  INSERT INTO unfollows` `  SET follower_id = OLD.follower_id,` `  followee_id = OLD.followee_id;` `END$$` `DELIMITER ;`	after a delete on follows we insert into another table called “unfollows”

Managing Triggers

SHOW TRIGGERS;	shows all the triggers in a database
DROP TRIGGER trigger_name;	deletes a trigger

- TRIGGERS can make debugging difficult because you don’t know if something is happening because of a trigger
- Be conscious of using TRIGGERS