CS 61:
Database Systems

Security

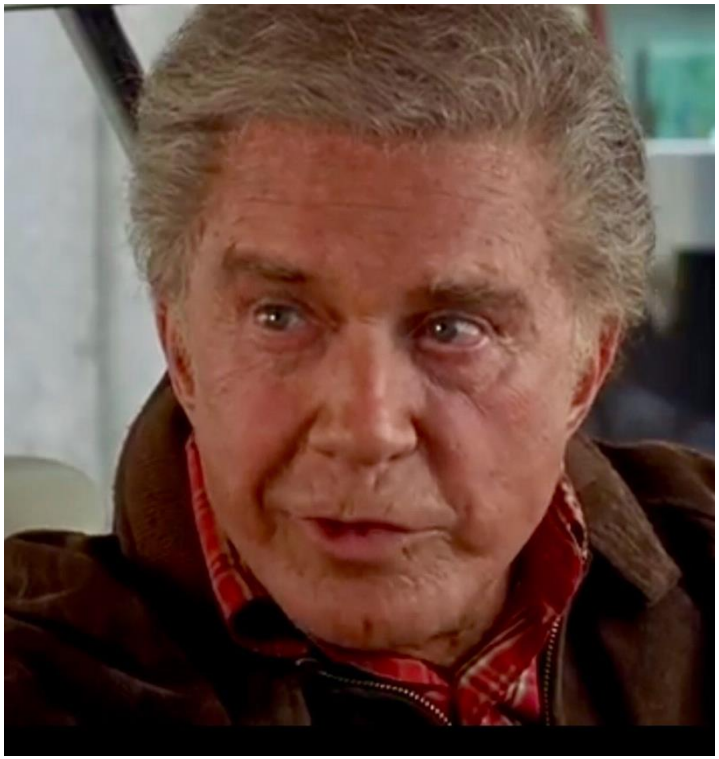
With great power comes great responsibility…

William Lamb, 2nd Viscount Melbourne

OR

Be wise!

You could get into trouble using some ideas from today!



Spider Man's uncle Ben

Source: Wikipedia

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Bcypt
5. JWT

SQL databases have a schema called INFORMATION_SCHEMA

INFORMATION_SCHEMA

- Virtual database that exists in every MySQL instance
- Provides *read-only* access to database metadata
  - Data about data
  - Describes all the databases, tables, columns, indexes, privileges, and other objects the server maintains
- Part of the ANSI SQL standard
  - Available in other databases
  - PostgreSQL, SQL Server, and other RDBMSs

Uses

- **Database discovery** — Explore an unfamiliar database without documentation
- **Schema validation** — Programmatically check that tables/columns exist before running code
- **Application development** — ERD tools rely on it behind the scenes
- **Security auditing** — See who has access to what database items
  - It respects privileges — Users only see metadata for objects they have permission to access

Many GUI database tools use INFORMATION_SCHEMA behind the scenes

Find all schemas by querying SCHEMATA

```
5     -- SLIDE 5
6 •   SET @schemaName = 'nyc_data';
7
8     -- find all schemas
9 •   SELECT SCHEMA_NAME
10        FROM INFORMATION_SCHEMA.SCHEMATA
11        ORDER BY SCHEMA_NAME;
12
```

00% 20:1

Result Grid | Filter Rows: Search | Export:

SCHEMA_NAME
classicmodels
college
dart_air
information_schema
lab2_db
lab3_db
midterm1
mysql
nyc_data
painting_gallery1
painting_gallery2
performance_sche…
projects
silbershatzbook
sql-murder-mystery
sys
test_schema

Find all tables in a schema by querying TABLES

```
15 •   SELECT TABLE_NAME, TABLE_TYPE, ENGINE, TABLE_ROWS, CREATE_TIME
16         FROM INFORMATION_SCHEMA.TABLES
17         WHERE TABLE_SCHEMA = @schemaName;
```

00% 8:15

Result Grid Filter Rows: Search Export:

TABLE_NAME	TABLE_TYPE	ENGINE	TABLE_ROWS	CREATE_TIME
Actions	BASE TABLE	InnoDB	5	2026-04-29 18:04:53
Cuisine	BASE TABLE	InnoDB	90	2026-04-29 18:04:53
Inspections	BASE TABLE	InnoDB	53046	2026-04-29 18:04:53
InspectionTypes	BASE TABLE	InnoDB	34	2026-04-29 18:04:53
InspectionViolations	BASE TABLE	InnoDB	130165	2026-04-29 18:04:53
restaurant_inspections	BASE TABLE	InnoDB	281096	2026-04-29 18:04:54
Restaurants	BASE TABLE	InnoDB	29565	2026-04-29 18:04:58
ViolationCodes	BASE TABLE	InnoDB	151	2026-04-29 18:04:59

Find all columns in a table by querying COLUMNS

```
20      -- Describe the structure of a table
21 •    SELECT COLUMN_NAME, DATA_TYPE, IS_NULLABLE,
22           COLUMN_DEFAULT, CHARACTER_MAXIMUM_LENGTH,
23           COLUMN_KEY, EXTRA
24         FROM INFORMATION_SCHEMA.COLUMNS
25         WHERE TABLE_SCHEMA = @schemaName
26             AND TABLE_NAME = 'Restaurants'
27         ORDER BY ORDINAL_POSITION;
28
```

100% 12:17

Result Grid | Filter Rows: Search | Export:

COLUMN_NAME	DATA_TYPE	IS_NULLABLE	COLUMN_DEFAULT	CHARACTER_MAXIMUM_LENGTH	COLUMN_KEY	EXTRA
RestaurantID	bigint	NO	NULL	NULL	PRI	
RestaurantName	varchar	YES	NULL	100		
Boro	varchar	YES	NULL	20		
Building	varchar	YES	NULL	20		
Street	varchar	YES	NULL	100		
ZipCode	int	YES	NULL	NULL		
Phone	bigint	YES	NULL	NULL		
Latitude	double	YES	NULL	NULL		
Longitude	double	YES	NULL	NULL		
InspectionAvg	float	YES	NULL	NULL		
InspectionCount	float	YES	NULL	NULL		
CuisineID	int	YES	NULL	NULL	MUL	

Find foreign key relationships by querying KEY_COLUMN_USAGE

```
31 •   SELECT TABLE_NAME, COLUMN_NAME,
32             CONSTRAINT_NAME, REFERENCED_TABLE_NAME, REFERENCED_COLUMN_NAME
33         FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
34         WHERE TABLE_SCHEMA = @schemaName
35             AND REFERENCED_TABLE_NAME IS NOT NULL;
36
```

100% 11:29

Result Grid Filter Rows: Search Export:

TABLE_NAME	COLUMN_NAME	CONSTRAINT_NAME	REFERENCED_TABLE_NAME	REFERENCED_COLUMN_NAME
Inspections	ActionID	fk_Inspections_Actions1	Actions	ActionID
Inspections	InspectionTypeID	fk_Inspections_InspectionTypes1	InspectionTypes	InspectionTypeID
Inspections	RestaurantID	fk_Inspections_Restaurants	Restaurants	RestaurantID
InspectionViolations	InspectionID	fk_InspectionViolations_Inspections1	Inspections	InspectionID
InspectionViolations	ViolationCode	fk_InspectionViolations_ViolationCodes1	ViolationCodes	ViolationCode
restaurant_inspections	CAMIS	restaurant_inspections_ibfk_1	Restaurants	RestaurantID
Restaurants	CuisineID	fk_Restaurant_Cuisine1	Cuisine	CuisineID

Use INFORMATION_SCHEMA to make a dictionary of all tables and columns

```
55    -- Generate a quick data dictionary
56 •  SELECT c.TABLE_NAME, c.COLUMN_NAME, c.DATA_TYPE,
57        c.COLUMN_KEY,c.IS_NULLABLE, c.COLUMN_DEFAULT, c.EXTRA
58        FROM INFORMATION_SCHEMA.COLUMNS c
59        WHERE c.TABLE_SCHEMA = @schemaName
60        ORDER BY c.TABLE_NAME, c.ORDINAL_POSITION;
61
```

100% 5:53

Result Grid Filter Rows: Search Export:

TABLE_NAME	COLUMN_NAME	DATA_TYPE	COLUMN_KEY	IS_NULLABLE	COLUMN_DEFAULT	EXTRA
Actions	ActionID	int	PRI	NO	NULL	auto_increment
Actions	ActionDescription	varchar		YES	NULL	
Cuisine	CuisineID	int	PRI	NO	NULL	auto_increment
Cuisine	CuisineDescription	varchar		YES	NULL	
Inspections	InspectionID	int	PRI	NO	NULL	auto_increment
Inspections	RestaurantID	bigint	MUL	NO	NULL	
Inspections	InspectionDate	date		YES	NULL	
Inspections	Score	int		YES	NULL	
Inspections	Grade	varchar		YES	NULL	
Inspections	GradeDate	date		YES	NULL	
Inspections	ActionID	int	MUL	YES	NULL	
Inspections	InspectionTypeID	int	MUL	YES	NULL	
InspectionTy...	InspectionTypeID	int	PRI	NO	NULL	auto_increment
InspectionTy...	InspectionTypeD...	varchar		YES	NULL	
InspectionVio...	InspectionID	int	PRI	NO	NULL	
InspectionVio...	ViolationCode	varchar	PRI	NO	NULL	
restaurant_in...	CAMIS	bigint	MUL	YES	NULL	
restaurant_in...	INSPECTION DA...	text		YES	NULL	
restaurant_in...	ACTION	text		YES	NULL	
restaurant_in...	VIOLATION CODE	text		YES	NULL	
restaurant_in...	VIOLATION DES...	text		YES	NULL	
restaurant_in...	CRITICAL FLAG	text		YES	NULL	
restaurant_in...	SCORE	int		YES	NULL	
restaurant_in...	GRADE	text		YES	NULL	
restaurant_in...	GRADE DATE	text		YES	NULL	
restaurant_in...	RECORD DATE	text		YES	NULL	
restaurant_in...	INSPECTION TY...	text		YES	NULL	
restaurant_in...	InspectionDate	date		YES	NULL	
restaurant_in...	GradeDate	date		YES	NULL	
restaurant_in...	RecordDate	date		YES	NULL	
restaurant_in...	InspectionYear	int		YES	NULL	VIRTUAL GE...
Restaurants	RestaurantID	bigint	PRI	NO	NULL	
Restaurants	RestaurantName	varchar		YES	NULL	
Restaurants	Boro	varchar		YES	NULL	

Create new database users, restricting where they can access the database

```
-- Can connect from ONLY the local machine
CREATE USER 'bob'@'localhost';

-- Can connect from ONLY this one IP
CREATE USER 'bob'@'192.168.1.50';

-- Can connect from ANY IP on the 192.168.1.x subnet
CREATE USER 'bob'@'192.168.1.%';

-- Can connect from ANY machine, anywhere
CREATE USER 'bob'@'%';
```

These users have no password!

MySQL will allow users without passwords!

Create new database users, restricting where they can access the database

```
-- Can connect from ONLY the local machine
CREATE USER 'bob'@'localhost' IDENTIFIED BY 'Bobs password';

-- Can connect from ONLY this one IP
CREATE USER 'bob'@'192.168.1.50' IDENTIFIED BY 'Bobs password';

-- Can connect from ANY IP on the 192.168.1.x subnet
CREATE USER 'bob'@'192.168.1.%' IDENTIFIED BY 'Bobs password'

-- Can connect from ANY machine, anywhere
CREATE USER 'bob'@'%' IDENTIFIED BY 'Bobs password'
```

Note:
bob@localhost is a different user than bob@%

Be careful!

Why not always use @%?

- **Security!**
- **Use localhost if only local access needed**
- **Use network mask if only accessible from a particular network (say at work)**
- **Be careful about granting @%, because anyone can connect (but they still need the password)**

Grant users the least privileges they need to get their job done!

```
82      -- create user bob
83  •   CREATE USER 'bob'@'localhost' IDENTIFIED BY 'password';
84
85      -- see bob's privileges
86  •   SELECT *
87          FROM INFORMATION_SCHEMA.USER_PRIVILEGES
88          WHERE GRANTEE LIKE '%bob%';
89
```

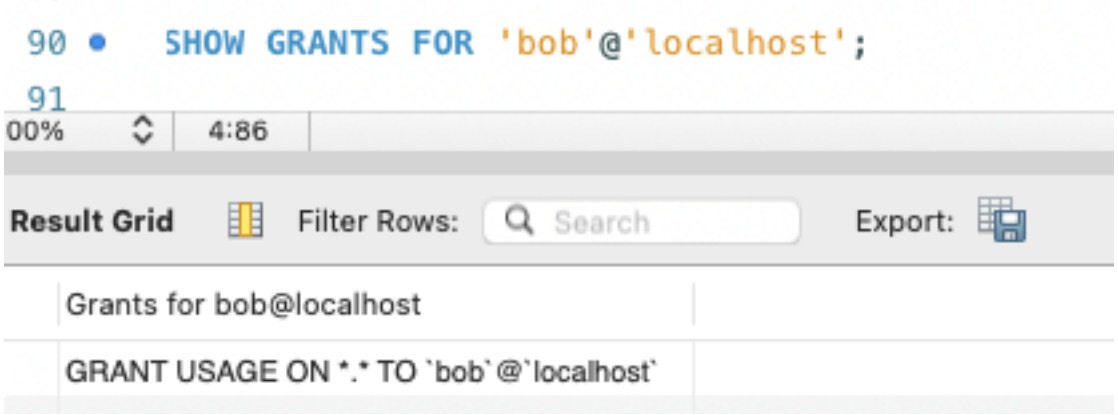
00% 23:78

Result Grid | Filter Rows: Search | Export:

GRANTEE	TABLE_CATALOG	PRIVILEGE_TYPE	IS_GRANTABLE
'bob'@'localhost'	def	USAGE	NO

Only usage privileges for bob
Usage means bob can log in, that is about it

Grant users the least privileges they need to get their job done!



Can also use
SHOW GRANTS FOR 'bob'@'localhost'

Grant users the least privileges they need to get their job done!

```
94      -- SLIDE 15
95      -- give bob all privileges!
96  •   GRANT ALL PRIVILEGES ON nyc_data.* TO 'bob'@'localhost';
97
98      -- see bob's privileges now
99  •   SHOW GRANTS FOR 'bob'@'localhost';
100
```

100% 1:92

Result Grid | Filter Rows: Search | Export:

Grants for bob@localhost
GRANT USAGE ON *.* TO `bob`@`localhost`
GRANT ALL PRIVILEGES ON `nyc_data`.* TO `...

Grant all privileges and bob can do everything

```
111     -- See schema-level privileges
112 •   SELECT *
113         FROM INFORMATION_SCHEMA.SCHEMA_PRIVILEGES
114         WHERE GRANTEE LIKE '%bob%';
115
```

100% 13:109

Result Grid | Filter Rows: Search | Export:

GRANTEE	TABLE_CATALOG	TABLE_SCHEMA	PRIVILEGE_TYPE	IS_GRANTABLE
'bob'@'localhost'	def	nyc_data	SELECT	NO
'bob'@'localhost'	def	nyc_data	INSERT	NO
'bob'@'localhost'	def	nyc_data	UPDATE	NO
'bob'@'localhost'	def	nyc_data	DELETE	NO
'bob'@'localhost'	def	nyc_data	CREATE	NO
'bob'@'localhost'	def	nyc_data	DROP	NO
'bob'@'localhost'	def	nyc_data	REFERENCES	NO
'bob'@'localhost'	def	nyc_data	INDEX	NO
'bob'@'localhost'	def	nyc_data	ALTER	NO
'bob'@'localhost'	def	nyc_data	CREATE TEMPORARY TABLES	NO
'bob'@'localhost'	def	nyc_data	LOCK TABLES	NO
'bob'@'localhost'	def	nyc_data	EXECUTE	NO
'bob'@'localhost'	def	nyc_data	CREATE VIEW	NO
'bob'@'localhost'	def	nyc_data	SHOW VIEW	NO
'bob'@'localhost'	def	nyc_data	CREATE ROUTINE	NO
'bob'@'localhost'	def	nyc_data	ALTER ROUTINE	NO
'bob'@'localhost'	def	nyc_data	EVENT	NO
'bob'@'localhost'	def	nyc_data	TRIGGER	NO

Bob has all privileges on nyc_data

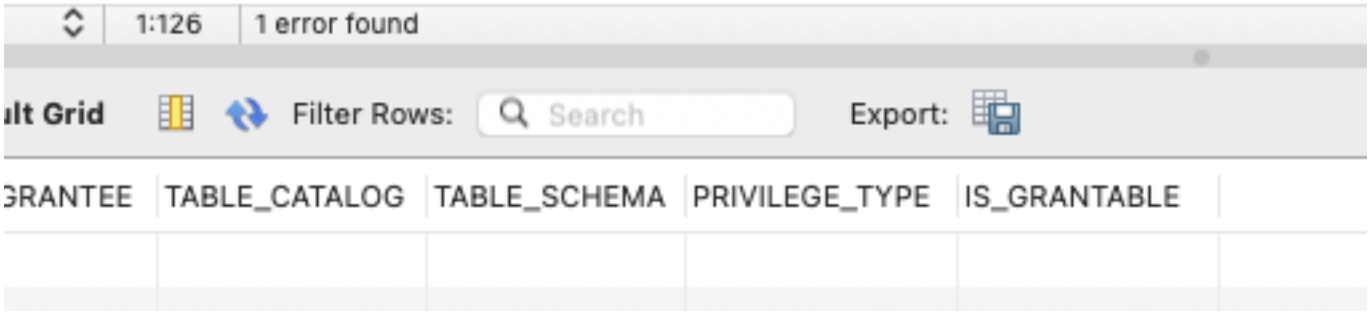
Except grant privileges to others
Use GRANT OPTION for that

Grant users the least privileges they need to get their job done!

```
-- revoke privileges
REVOKE ALL PRIVILEGES ON nyc_data.* FROM 'bob'@'localhost';

SHOW GRANTS FOR 'bob'@'localhost';
SELECT *
    FROM INFORMATION_SCHEMA.SCHEMA_PRIVILEGES
    WHERE GRANTEE LIKE '%bob%';
```

Revoke all privileges
(MySQL Workbench flags error, but works)



Common privileges

Privilege	Allows
`SELECT`	Read data
`INSERT`	Add rows
`UPDATE`	Modify rows
`DELETE`	Remove rows
`CREATE`	Create tables/databases
`DROP`	Drop tables/databases
`ALTER`	Alter table structure
`INDEX`	Create/drop indexes
`EXECUTE`	Run stored procedures
`GRANT OPTION`	Grant privileges to others
`ALL PRIVILEGES`	Everything above (and more)

Privileges can be granted on a schema or a specific table

Make API user from last class to access the database with minimal privileges

We are done with Bob

Drop to remove user

Make a user for the API to access database

```
129 •   DROP USER 'bob'@'localhost';
130
131     -- make user for API
132 •   CREATE USER 'webuser'@'localhost' IDENTIFIED BY 'password';
133 •   GRANT SELECT, INSERT ON nyc_data.Restaurants TO 'webuser'@'localhost';
134 •   GRANT SELECT, INSERT, UPDATE, DELETE ON nyc_data.Inspections TO 'webuser'@'localhost';
135 •   SHOW GRANTS FOR 'webuser'@'localhost';
136
```

Grant only necessary privileges by table

100% 1:124 1 error found

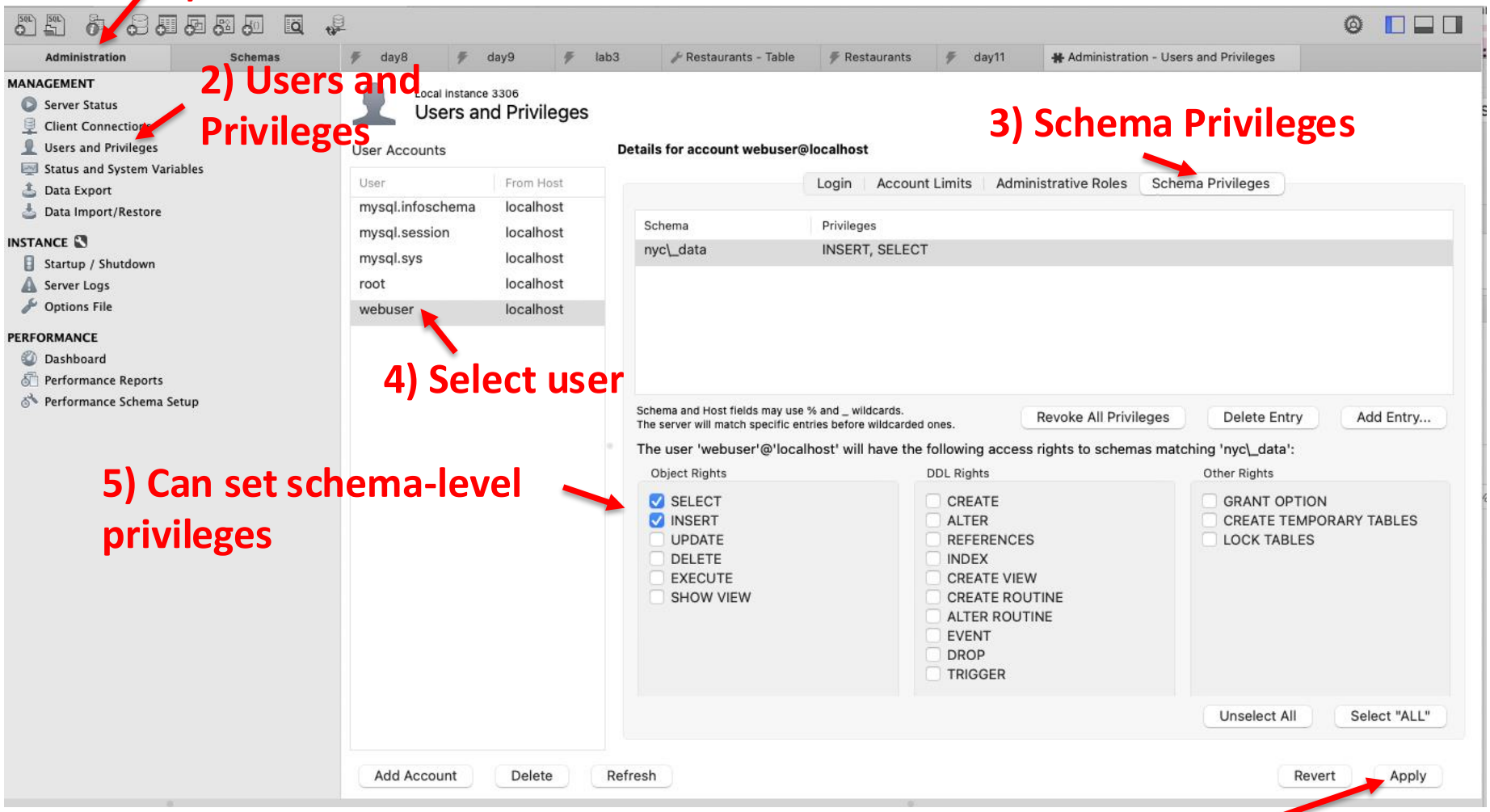
Result Grid Filter Rows: Search Export:

Grants for webuser@localhost
GRANT USAGE ON *.* TO `webuser`@`localhost`
GRANT SELECT, INSERT, UPDATE, DELETE ON `nyc_data`.`inspections` TO `webuser`@`localhost`
GRANT SELECT, INSERT ON `nyc_data`.`restaurants` TO `webuser`@`localhost`

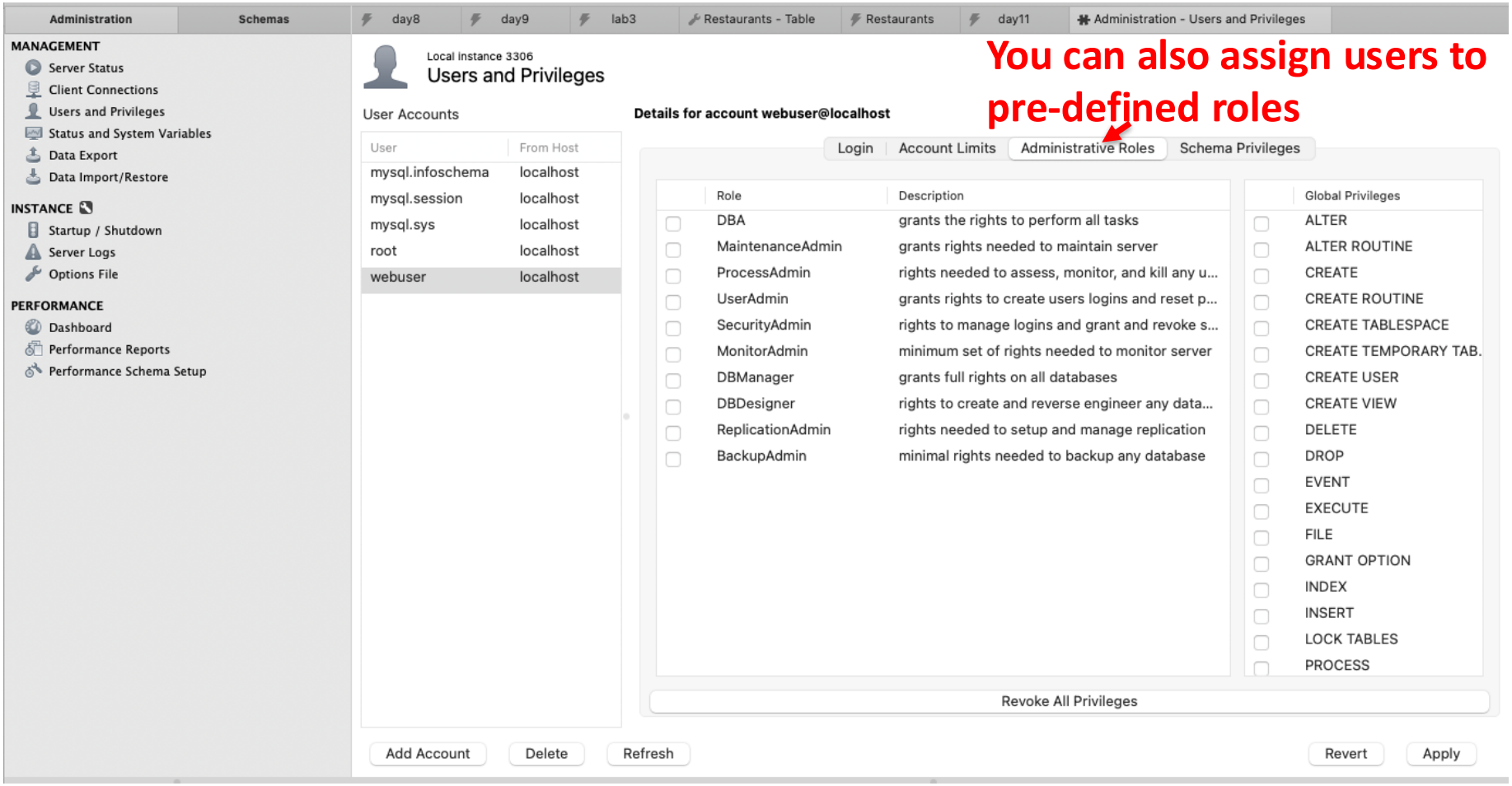
Update credentials in db.json

```
{
    "user":"webuser",
    "password":"password",
    "host":"some_server_name.cs.dartmouth.edu",
    "database":"nyc_data"
}
```

Alternatively, you can use the GUI in MySQL Workbench



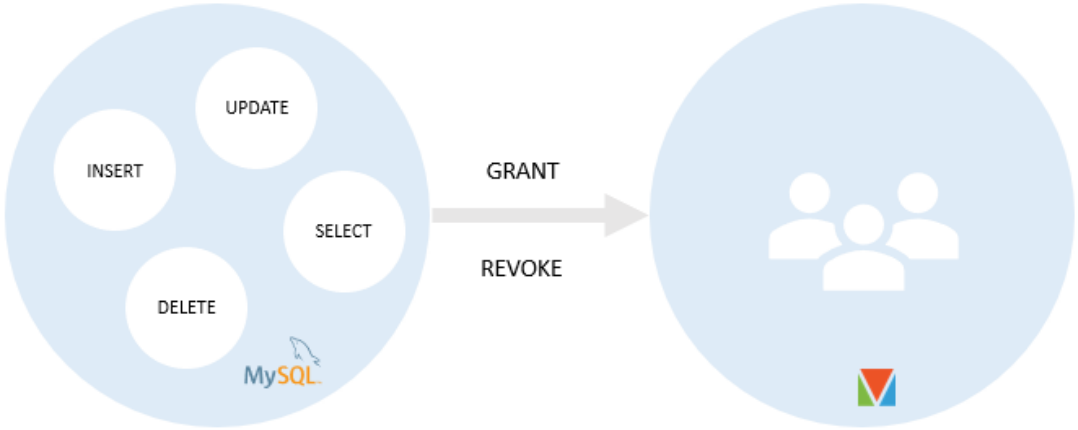
Alternatively, you can use the GUI in MySQL Workbench



Can assign rights to users individually or by role

Security authorization

Can assign rights to individual users



Can create roles, assign rights to roles, then assign users to roles

Benefits:

- Improved operational efficiency – new hires automatically get the rights they need
- Increased security – people do not get more rights that would typically need
- Increased visibility – easy to see what rights roles have

RBAC: Good idea in principle but has never worked for me!

- There is no generic person, each person has different responsibilities within dept
- People get temporary assignments with other departments, need different rights (creates a hybrid role)
- Assignment ends, but rights never changed (even if you set a calendar reminder and ask them if they still need the rights, they never say no!)

Adapted from: https://www.mysqltutorial.org/mysql-roles/

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Bcypt
5. JWT

Do not trust user input

Consider the following Python code making a SQL call for restaurant details

What is wrong with this Python code?

Hint: CONCAT is ok, it combines attributes together

```
restaurant = “nobu” #user input from textbox, Nobu is a restaurant
cursor = cnx.cursor()
query = ("SELECT RestaurantName AS `Restaurant Name`,"
        +"CONCAT(TRIM(Building),' ',TRIM(Street)) AS Address, "
        + "Boro  "
        +"FROM Restaurants "
        +"WHERE RestaurantName LIKE '%" + restaurant +"%’) "
        +"LIMIT 20"
cursor.execute(query)
return cursor
```

Nothing is wrong with this query, provided we can trust the value in restaurant

Adding user input directly into command is a recipe for trouble!

What is wrong with this Python code?

Hint: CONCAT is ok, it combines attributes together

```
restaurant = "nobu%' UNION SELECT 1,2,3 -- "
cursor = cnx.cursor()
query = ("SELECT RestaurantName AS `Restaurant Name`,"
         +"CONCAT(TRIM(Building),' ',TRIM(Street)) AS Address, "
         + "Boro  "
         +"FROM Restaurants "
         +"WHERE RestaurantName LIKE '%" + restaurant +"%')
         +"LIMIT 20"
cursor.execute(query)
return cursor
```

What if the user enters this instead? Note: space at end

UNION adds rows from the following SELECT (number of attributes must match in each query)
LIMIT is commented out as a result of user input

Query is now:

… WHERE RestaurantName LIKE '%nobu%' UNION SELECT 1,2,3 -- LIMIT 20

sql_injection.py demonstrates injection vulnerabilities

-- test if user entry is vulnerable to injection, should see extra row with 1,2,3 if so
nobu%' UNION SELECT 1,2,3 --

-- find out what schemas are on this database installation (see nyc_data)
nobu%' UNION SELECT schema_name, null, null from
information_schema.schemata --

-- find tables in nyc_data schema (see Users table)
nobu%' UNION SELECT table_name, table_schema, null from
information_schema.tables where table_schema = 'nyc_data' --

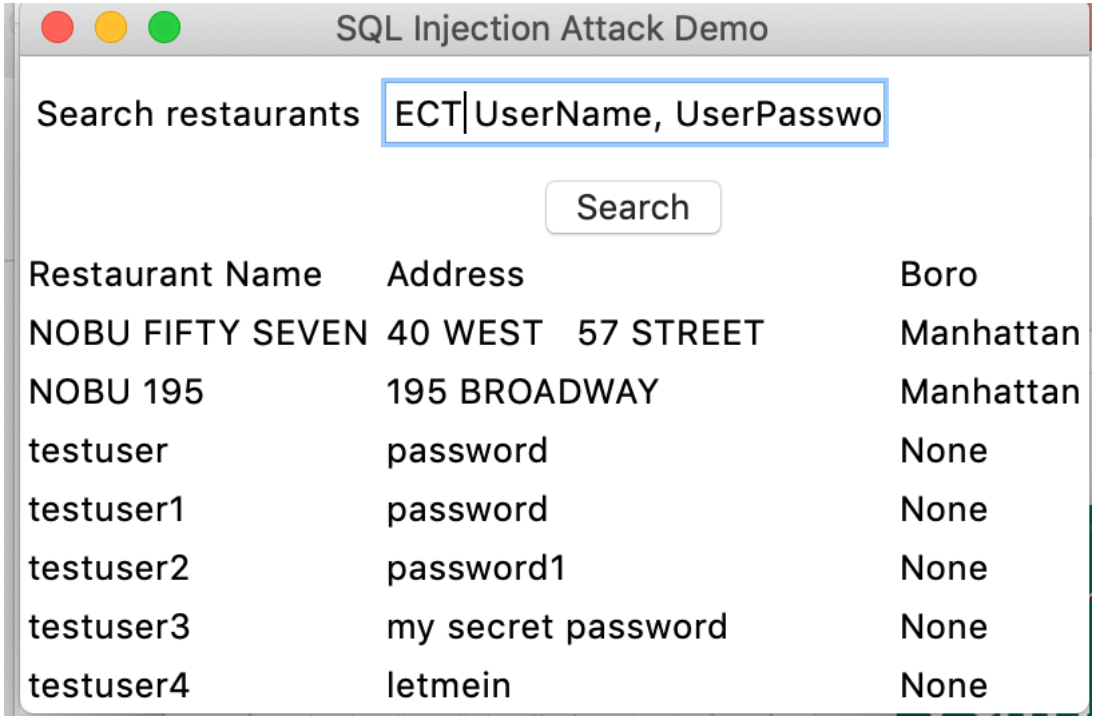
-- find attributes for Users table in nyc_data schema (see UserName and Password)
nobu%' UNION (SELECT column_name, data_type, character_maximum_length from
information_schema.columns where table_schema = 'nyc_data' and table_name =
'Users') –

-- Get UserNames and Passwords (Pwned!)
nobu%' UNION SELECT UserName, Password, null from nyc_data.Users --

Most sites have a Users table, let's steal all the usernames and passwords

-- I've created a Users table in nyc_data
-- let's steal the username and passwords of all users!

nobu%' UNION SELECT UserName, Password, null from nyc_data.Users --



You've been pwned!

We have at least two problems:

1. **Prevent SQL injection**
2. **Store hashed passwords**

Do not store passwords in plain text!

More on this soon

Use prepared statement to avoid user input as part of SQL command

Vulnerable	Prepared statement
restaurant = “nobu” cursor = cnx.cursor() query = ("SELECT RestaurantName, " +"Building, " + "Boro " +"FROM Restaurants " +"WHERE RestaurantName LIKE" **+"'%" + restaurant +"%’) "** +"LIMIT 20”) **cursor.execute(query)** return cursor	restaurant = “nobu” cursor = cnx.cursor() query = ("SELECT RestaurantName, " +"Building, ", +"Boro, " +"FROM Restaurants " +"WHERE RestaurantName LIKE” **+" %s ”** +"LIMIT 20") **cursor.execute(query, ('%'+restaurant+'%’,))** return cursor

User input is included in the SQL query string

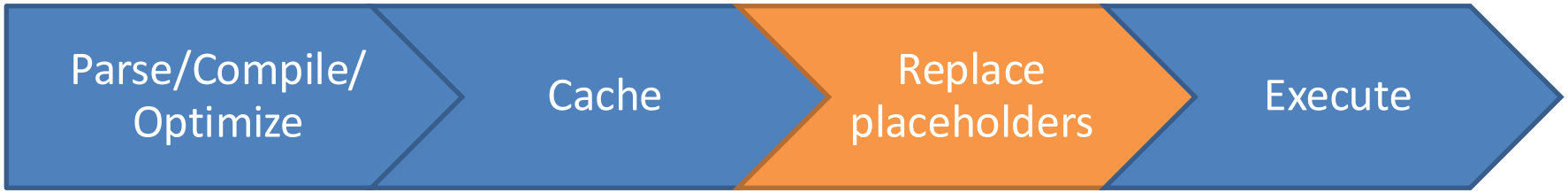
- **Can be abused!**

Prepared statements (sometimes called parameterized queries) add user input as a parameter after command is compiled

Prepared statements add data after compiling, optimizing, and caching

High-level overview of SQL execution process

UPDATE Users SET UserName = %s AND Password = %s



Parse
- Check syntax
- Check table and columns exist

Compile
- Convert query to machine code

Optimize
- Choose optimal execution plan

Cache
- Store optimized query plan in cache
- If command submitted again, skip prior steps (already done)

Replace placeholders
- Prepared statement are not complete statements
- Have placeholders for some values
- But, format of command is set now
- Placeholders filled with literal values
- Place holder data doesn't change command format

Execute
- Query is executed
- Data is returned
- Malicious command are stored in table as text, not executed

User input is just treated as text, NEVER part of the SQL command

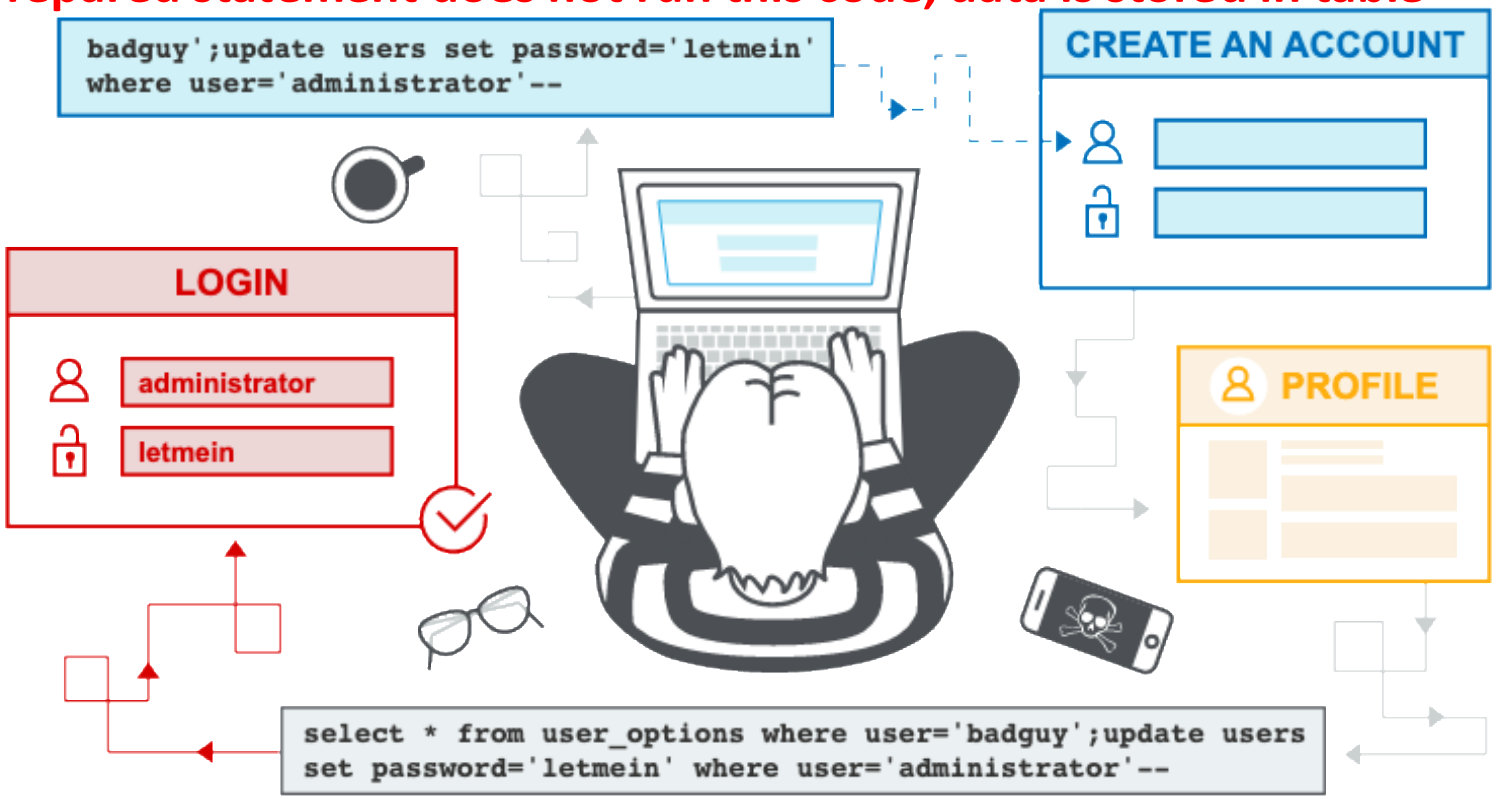
Input: Admin' -- Quote is escaped

Adapted from: http://javabypatel.blogspot.com/2015/09/how-prepared-statement-in-java-prevents-sql-injection.html

Even if you use prepared statements, be wary of data in your database!

Second-order attack:
User enters data with SQL embedded
Prepared statement does not run this code, data is stored in table

Called a stored injection attack



Can't trust data in database either!

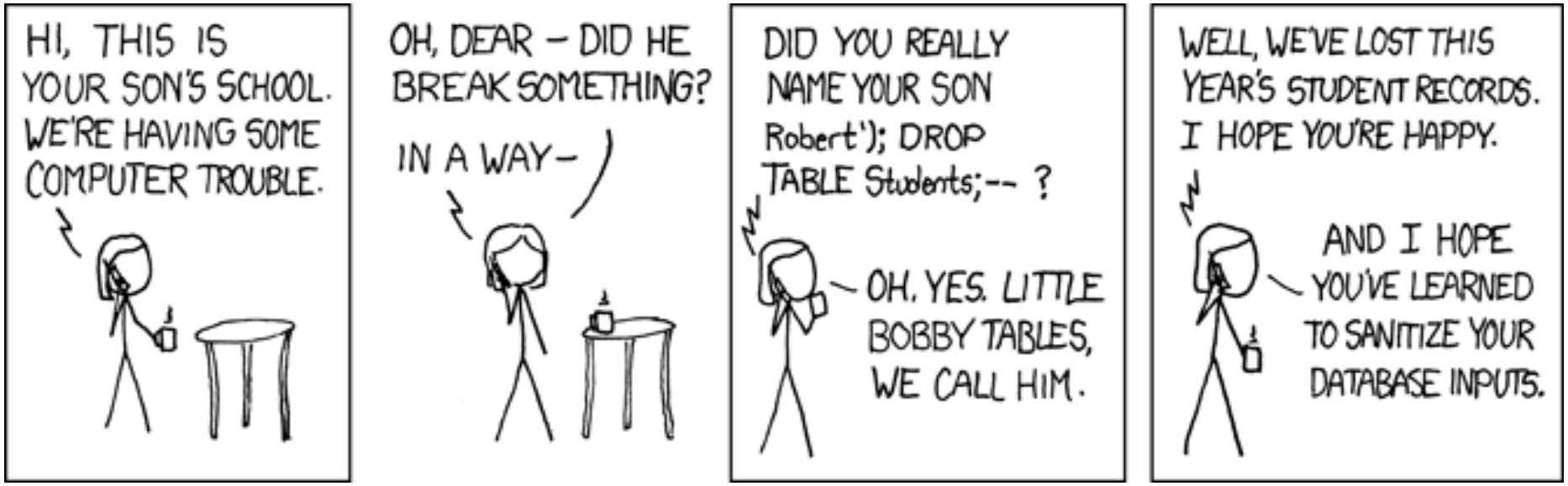
Set minimal privileges!

Attack at the Met

Later someone runs a command where user = 'badguy'
Programmer assumes data in database is safe
Command executes; here resets admin password

https://portswigger.net/web-security/sql-injection

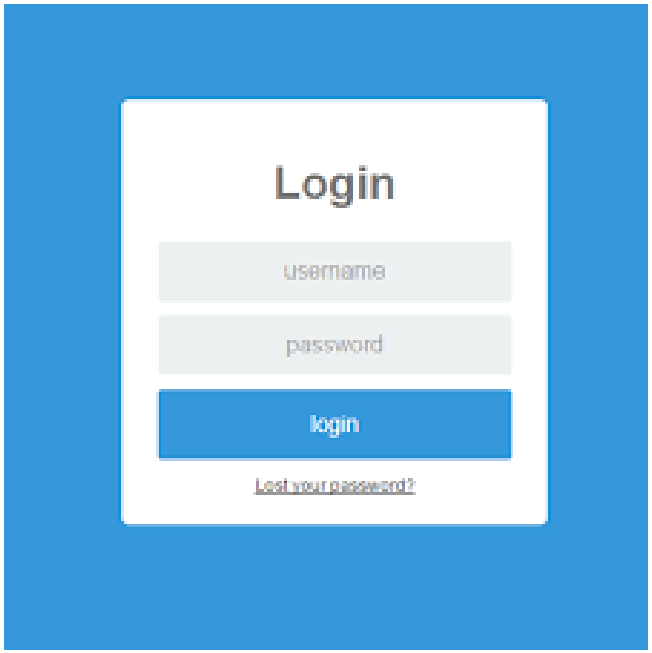
Now we know why the comic on the course web site is funny!



Source: https://xkcd.com/327/

Practice

Assume a log in form issues the following SQL behind the scenes where user input is used directly in the SQL:



Enter: administrator' --
Command now:
SELECT * **FROM** Users **WHERE** UserName = 'administrator'--
AND Password = 'password'

SELECT * **FROM** Users **WHERE** UserName = 'username' **AND** Password = 'password'

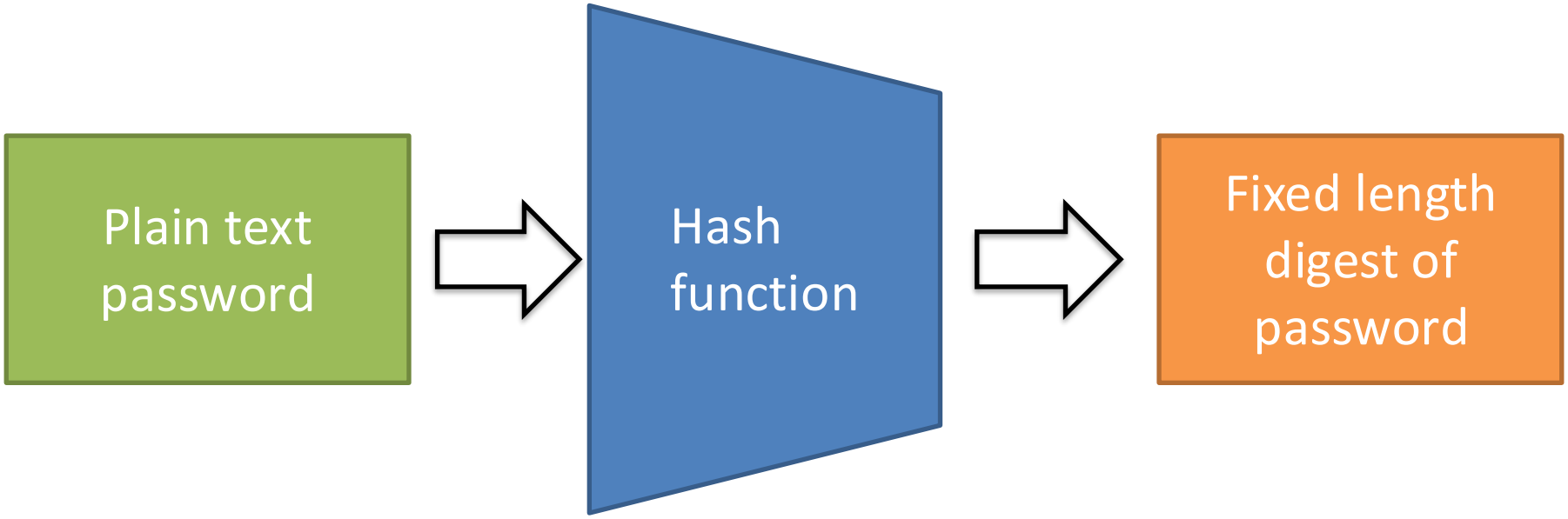
The site then logs you in if one row is returned by the query

What could you enter in the username or password fields to log in as 'administrator' even if you do not know the password?

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Bcypt
5. JWT

Review: hashing takes plain text and outputs a fixed-length digest



Input:
"my secret password"

Output:
a7303f3eee5f3ff1942bfbb1797ea0af

Hash function is a deterministic mathematical one-way trap door

- **Cannot find plain text in "reasonable" amount of time given only the hash digest**
- **Or can we?**

DO NOT store user passwords in plain text!

UserID	UserName	UserPassword
1	testuser	password
2	testuser1	password
3	testuser2	password1
4	testuser3	my secret password

Do not store passwords in plain text

Note: same password results in same hash

If adversary steals passwords, cannot read plain-text password

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Instead store hash of password

On log in: hash plain text password and compare with database

Username: “testuser”
Password: “password”

Hash user’s plain text password and look for match in database for this username

Because hash function is deterministic, same password will always result in same digest

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Hashes match for testuser

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

User submitted valid password

Dictionary attack: try all words in a dictionary looking for a match

Username: “testuser”
Password: “password”

Hash Password

Change your password if in dictionary!

Right now!

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Assume adversary steals hashed passwords

Password: “aardvark”
Password: “alice”
Password: “anteater”
…
Password: “password”

Dictionary attack:
Hash all words in a dictionary, if word hash matches database hash, password is “cracked”

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Will not crack if user’s password not in dictionary

Crack one, crack all with same password

Rainbow table attacks precompute all possible character combinations

Username: “testuser”
Password: “password”

Hash Password

Hashed password:
5f4dcc3b5aa765d61d8327deb882cf99

Password: “a”
Password: “aa”
Password: “aaa”
…
Password: “password”

Assume adversary steals hashed passwords

Rainbow table attack:
Precompute all character combinations up to certain length
Store resulting hash for each combo

UserID	UserName	HashedPassword
1	testuser	5f4dcc3b5aa765d61d8327deb882cf99
2	testuser1	5f4dcc3b5aa765d61d8327deb882cf99
3	testuser2	7c6a180b36896a0a8c02787eeafb0e4c
4	testuser3	a7303f3eee5f3ff1942bfbb1797ea0af

Look up database password in rainbow table

Lots of time and storage needed

Length limited

Use salt to prevent attacks

Username: “testuser”
Password: “password”

Password + **Salt**: “password**.ef_ob’3**”
Salted hashed password:
62c21dd30b2d7e6e6671628458aeaf1f

Salt:

- **Random string of characters appended (or prepended or both) to password before hash**
- **Each user gets unique salt**
- **Salt stored in plain text in database**
- **User need not know value of salt, it is added on server side**

If salt is long (say 64 characters) rainbow table is impractical

Dictionary attack still possible

- **Password plus salt unlikely to be in database**
- **Slows adversary by adding salt to each pwd**

UserID	UserName	Salt	SaltedPassword
1	testuser	.ef_ob'3	62c21dd30b2d7e6e6671628458aeaf1f
2	testuser1	\s#>2!x}	a9055805cb7e588dc27945cb95067f6b
3	testuser2	as=8KlA=	19e3385ed6bfe321b36b6bc4290bea0b
4	testuser3	n%lzA7QQ	a6fc8f715df44839b50cf31b639b961c

Adding pepper is even better

Username: “testuser”
Password: “password”

Password

Password + Salt **+ Pepper**: “password.ef_ob’3**Secret**”
Salted hashed password:
2811922850bbcd79683b58e43d1ab76f

Pepper:

- **Random string of characters appended to password + salt before hash**
- **Pepper kept secret, not stored in database**
- **One pepper for all users**

Another variant

- **Pepper is one character chosen at random for each user**
- **Not stored**
- **On log, try ‘a’, then ‘b’**
  - **Will eventually find match**
  - **Slows adversary**

UserID	UserName	Salt	SaltedPassword
1	testuser	.ef_ob'3	62c21dd30b2d7e6e6671628458aeaf1f
2	testuser1	\s#>2!x}	a9055805cb7e588dc27945cb95067f6b
3	testuser2	as=8KlA=	19e3385ed6bfe321b36b6bc4290bea0b
4	testuser3	n%lzA7QQ	a6fc8f715df44839b50cf31b639b961c

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Bcypt
5. JWT

Hash passwords: bcrypt is a popular hashing library

Each hash is deliberately slow to compute (around 300 ms or so)

- Uses many rounds of hashing to increase the cost to compute hash
- The number of rounds can be increased as computation power increases
- Why is a slow hashing function good?
  - Makes dictionary/rainbow table attacks difficult
  - Users will not notice a short delay in computing hash

Generates salt

- Makes dictionary/rainbow table attacks difficult

Hashed password includes the salt

- One column is needed in the database for the password and salt

Hash passwords: bcrypt is a popular hashing library

```
password = "test"
salt = bcrypt.gensalt()
hash = bcrypt.hashpw(password.encode('UTF-8'),salt)

print(f'Password: {password}')
print(f'Salt: {salt}')
print(f'Hashed password: {hash.decode('UTF-8')}')

#test decoding
print(bcrypt.checkpw(password.encode('UTF-8'),hash))
```

Generate salt

Hash with salt

Convert password to bytes

```
Password: test
Salt: b'$2b$12$xDjm88WAxyWPQ92iZexHbO'
Hashed password: $2b$12$xDjm88WAxyWPQ92iZexHbOW2vPdLI2FY30YtD2YwYLAFdpZ3DeUsW
True
```

Salt

Bcrypt version

12 rounds of hashing

Hashed and salted password

Store *hash* (with salt) in database

Store the hashed password (with salt) in the database

For demonstration only
DO NOT store the plaintext password!

Store the salted and hashed password instead

UserID	UserName	Password	HashedPassword
1	testuser	password	$2b$12$qeynkbIoOCAlg8KVNkj2c.62KGmfn..4enKzPim8pJ5oQPL6n5tB.
2	testuser1	password	$2b$12$GN/5vEpSFR80yTvgkEcctubGsSs/i1MrP9idU7mHCF7S2HDb6aaIC
3	testuser2	password1	$2b$12$tt2vGiVEyOuHqDbvHDKHhOQW38t5rTG3GecXC4pk0KozjBirEKTR2
4	testuser3	my secret password	$2b$12$YyBbvBQ3ePHvHZuNuKucceWMIA.feozF5bHmVAkQJjC/04lj4YDmO
5	testuser4	letmein	$2b$12$OvehyOHiMh3RF8D9iqtEpuj3PvKcAqqEriDEzYYr8pkay28x.ceJ.

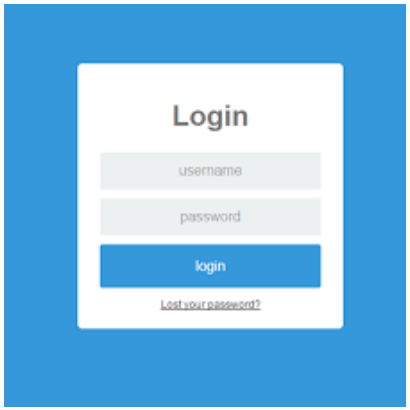
Notice that with salt, users who have the same password get a different hash

testuser **and** ***testuser1*** **both have the same password but different hash**

Cracking one password does not crack both

Check hash with bcrypt

Client side



- User enters username and password
- Clicks login button
- Code calls /login route passing username and password in JSON

Server side /login route

```python
@app.route('/api/login', methods=['POST'])
def login():
    data = request.get_json()

    if not data or 'UserName' not in data or 'Password' not in data:
        return jsonify({"error": "Email and password are required"}), 400

    db = get_db_connection()
    cursor = db.cursor(dictionary=True)

    # Get employee data for the UserName
    # Use a parameterized query!
    # Store the result in a variable called 'employee'
    cursor.execute("SELECT * FROM Employees WHERE UserName = %s", (data['UserName'],))
    employee = cursor.fetchone()

    cursor.close()
    db.close()

    if not employee:
        return jsonify({"error": "Invalid username"}), 401

    # Compare data['password'] against employee['password_hash']
    # Remember: both need to be bytes, so use .encode('utf-8')
    password_valid = bcrypt.checkpw(data['Password'].encode('utf-8'),
        employee['PasswordHash'].encode('utf-8'))
```

Get JSON passed in

Check got both username and password

Use *parameterized* query to get this user's hashed password

Invalid if no user found

***bcrypt.checkpw* hashes plaintext password with salt, then checks result matches stored salted and hashed password**

Agenda

1. MySQL permissions
2. Demo: SQL injection attacks
3. Password storage/salt and pepper
4. Bcypt
5. JWT

We need a method to authenticate users on subsequent endpoint calls

/login route expects users to provide username and password
Route hashes plaintext password with salt from database and compares with stored hash:

- If match, authorized user, grant access to other routes
- Else, unauthorized user, do no grant access to other routes

Two questions:

1. How do other routes know if the user has already authenticated?

2. Should all users be able to call all endpoints?
   - Can all users call DELETE /restaurants
   - Maybe only admins should be able to delete restaurants

Option 1: send username and password in headers on EVERY request

Client sends username and password with EVERY request, encoded in Base64:

GET /employees
Headers:
Authorization: Basic YWRtaW46cGFzc3dvcmQxMjM=

This is just "admin password123" in Base64, passed by client on each endpoint call

```
response = requests.get(url, auth=('admin password123'))
```

```python
import base64

@app.route('/employees', methods=['GET'])
def get_employees():
    auth_header = request.headers.get('Authorization')

    if not auth_header or not auth_header.startswith('Basic '):
        return jsonify({"error": "Authentication required"}), 401

    # Decode the Base64 string
    encoded = auth_header.split(' ')[1]
    decoded = base64.b64decode(encoded).decode('utf-8')
    username, password = decoded.split(':')

    # Verify against database
    # ...
```

Caller set Basic Authentication in request header

Check for Basic Authentication

Recover username/password sent in every request

claude.ai

Option 1: Pros and cons

Pros

- ✅ Very simple — built into HTTP standard
- ✅ Supported by every HTTP client, browser, and tool
- ✅ No token management needed

Cons

- ❌ Password sent with EVERY request (higher exposure risk)
- ❌ Must use HTTPS — completely insecure without it
- ❌ No expiration — valid as long as password is valid
- ❌ Server must verify password every single time (slower)
- ❌ No way to revoke a session without changing the password

Option 2: Session/cookie-based authentication

Step 1: User logs in

POST /login {"username": "alice", "password": "secret"}

Step 2: Server creates a session and sends back a cookie

Response Headers:

Set-Cookie: session_id=abc123xyz; HttpOnly; Secure

Step 3: Browser automatically sends cookie with every future request

GET /employees

Headers:

Cookie: session_id=abc123xyz

Step 4: Server looks up session_id in memory/database to find the user

Option 2: Set session/cookie-based variables on successful login

```python
from flask import Flask, session, request, jsonify
import secrets

app = Flask(__name__)
app.secret_key = 'your-secret-key'

@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()

    # Verify credentials against database
    db = get_db()
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT * FROM employees WHERE email = %s", (data['email'],))
    user = cursor.fetchone()
    cursor.close()
    db.close()

    if not user:
        return jsonify({"error": "Invalid credentials"}), 401

    # Now 'user' is defined and we can use it
    session['user_id'] = user['id']
    session['is_admin'] = user['is_admin']

    return jsonify({"message": "Logged in"}), 200
```

Client sends to /login
{“username”:”username”
”password”:”secret”}

Server checks against database

Return error if unsuccessful login

Set session variable on successful login

claude.ai

Option 2: Other routes check session variable

```python
@app.route('/restaurants', methods=['GET'])
def get_restaurants():
    if 'user_id' not in session:
        return jsonify({"error": "Please log in"}), 401

    # User is authenticated
    user_id = session['user_id']
    # ...

@app.route('/logout', methods=['POST'])
def logout():
    session.clear()
    return jsonify({"message": "Logged out"}), 200
```

Check session variable in other routes

Clear session variable on logout

claude.ai

Option 2: Pros and cons

Pros

- ✅ Browser handles cookies automatically — no client-side code needed
- ✅ Easy to invalidate (delete the session on the server)
- ✅ Server has full control (can end any session at any time)
- ✅ Well understood, battle-tested for decades

Cons

- ❌ Requires server-side storage (memory, database, or Redis)
- ❌ Doesn't scale well — session must be shared across multiple servers
- ❌ Vulnerable to CSRF attacks (need extra protection)
- ❌ Doesn't work well for mobile apps or third-party API clients
- ❌ Tied to a single domain (cookies don't cross domains easily)

Option 3: Use JSON Web Token

Step 1: User logs in

POST /login {"email": "alice@co.com", "password": "secret"}

Step 2: Server creates a signed token and sends it back

Response: {"token": "eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxfQ.abc123"}

Step 3: Client stores the token and sends it with every request

GET /employees

Headers:

Authorization: Bearer eyJhbGciOiJIUzI1NiJ9...

Step 4: Server verifies the token signature (no additional database lookups)

You will do this on Lab 4

Option 3: Pros and cons

Pros

- ✅ Stateless — server doesn't need to store sessions
- ✅ Scales easily — any server can verify the token
- ✅ Works for web, mobile, desktop, server-to-server
- ✅ Token contains user info (no database lookup to identify user)
- ✅ Built-in expiration

Cons

- ❌ Can't easily revoke a token before it expires
- ❌ Token size is larger than a session ID
- ❌ If stolen, valid until expiration
- ❌ Client must manage token storage
- ❌ More complex to implement correctly

Add Admin column on database to allow some users to access sensitive endpoints

UserID	UserName	Admin	Password	HashedPassword
1	testuser	1	password	$2b$12$qeynkbIoOCAlg8KVNkj2c.62KGmfn..4enKzPim8pJ5oQPL6n5tB.
2	testuser1	0	password	$2b$12$GN/5vEpSFR80yTvgkEcctubGsSs/i1MrP9idU7mHCF7S2HDb6aaIC
3	testuser2	1	password1	$2b$12$tt2vGiVEyOuHqDbvHDKHhOQW38t5rTG3GecXC4pk0KozjBirEKTR2
4	testuser3	0	my secret password	$2b$12$YyBbvBQ3ePHvHZuNuKucceWMIA.feozF5bHmVAkQJjC/04lj4YDmO
5	testuser4	0	letmein	$2b$12$OvehyOHiMh3RF8D9iqtEpuj3PvKcAqqEriDEzYYr8pkay28x.ceJ.

Admin users have more rights

Can access sensitive endpoints

For example: only admins should be able to delete a restaurant

Add wrapper to only allow logged in admins to access some endpoints

```python
def token_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None

        # JWT token should be in the request header as:
        # Authorization: Bearer <token>
        if 'Authorization' in request.headers:
            auth_header = request.headers['Authorization']
            parts = auth_header.split()
            if len(parts) == 2 and parts[0] == 'Bearer':
                token = parts[1]

        if not token:
            return jsonify({"error": "Token is missing. Please log in."}), 401

        try:
            # Decode the token using our secret key
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms=["HS256"])
            current_user_id = data['EmployeeID']
            current_user_is_admin = data['Admin']
        except jwt.ExpiredSignatureError:
            return jsonify({"error": "Token has expired. Please log in again."}), 401
        except jwt.InvalidTokenError:
            return jsonify({"error": "Token is invalid."}), 401

        # Pass the user info to the route function
        return f(current_user_id, current_user_is_admin, *args, **kwargs)

    return decorated
```

Make sure users have a valid login token

Use as a wrapper around other routes

Add wrapper to only allow logged in admins to access some endpoints

```python
# Similarly, this decorator checks that the user is an admin
def admin_required(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        token = None

        if 'Authorization' in request.headers:
            auth_header = request.headers['Authorization']
            parts = auth_header.split()
            if len(parts) == 2 and parts[0] == 'Bearer':
                token = parts[1]

        if not token:
            return jsonify({"error": "Token is missing. Please log in."}), 401

        try:
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms=["HS256"])
            current_user_id = data['EmployeeID']
            current_user_is_admin = data['Admin']
        except jwt.ExpiredSignatureError:
            return jsonify({"error": "Token has expired. Please log in again."}), 401
        except jwt.InvalidTokenError:
            return jsonify({"error": "Token is invalid."}), 401

        # Check if user is admin
        if not current_user_is_admin:
            return jsonify({"error": "Admin access required."}), 403

        return f(current_user_id, current_user_is_admin, *args, **kwargs)

    return decorated
```

Make sure users are admins

Use as a wrapper around other routes

Use wrapper to protect routes from unauthorized users

```
#get a single restaurant by ID as a parameter
@app.route('/restaurants', methods=['GET'])
@token_required
@admin_required
def get_restaurant():
    try:
        cnx = get_db_connection()
        cursor = cnx.cursor(prepared=True)  # use a prepared statement to avoid SQL injection!
        query = ("SELECT RestaurantID, RestaurantName, Boro, CuisineDescription, InspectionAvg "
                 "FROM Restaurants r JOIN Cuisine c USING (CuisineID) "
                 "WHERE RestaurantID = %s")
        cursor.execute(query, (id,))
        rows = cursor.fetchall()

        return jsonify(rows), 200 #status 200 = OK
```

Add decorators so only logged in admins can access this route

Other options

- OAuth 2.0 (use third party like Google to authenticate)
- mTLS (mutual TLS)