# Databases

# Databases

- Software that stores data on disk

- Runs as a server and is communicated with via TCP sockets

- Provides an API to store/retrieve data

  - The software handles the low-level file IO

  - Allows us to think about our data, not how to store it

- Provides many optimizations

# Databases

- We'll look at 2 different database

- Both are pieces of software that must be downloaded, installed, ran, then connected to via TCP

- mySQL

  - A server implementing SQL (Structured Query Language)

- MongoDB

  - A server based on document stores

# MySQL

- One you download, install, and run the server

  - It will listen for TCP connections on port 3306 (By default)

- Install a library for your language that will connect to the MySQL server

  - You will not have to connect to your database at the TCP level in this course (True for MongoDB as well)

  - The library will provide a convenient API

    - Send queries using the query language

# MySQL

- After MySQL is running and you install a library to connect to it

- Connect to MySQL Server by providing

  - The url of the database

  - username/password for the database

    - Whatever you chose when setting up the database

```
val url = "jdbc:mysql://localhost/mysql"
val username = "root"
val password = "12345678"

var connection: Connection = DriverManager.getConnection(url, username, password)
```

# MySQL - Security

- **For real apps that you deploy**

  - **Do not check your password into version control!**

    - **A plain text password in public GitHub repo is bad**

    - **Attacker can replace localhost with the IP for your app and can access all your data**

  - **Common to save the password in a environment variable to prevent accidentally pushing it to git**

  - **Do not use the default password for any servers you're running**

    - **This is what caused the Equifax leak (Not with MySQL)**

- **Attacker have bots that scan random IPs for such vulnerabilities**

```
val url = "jdbc:mysql://localhost/mysql?serverTimezone=UTC"
val username = "root"
val password = "12345678"

var connection: Connection = DriverManager.getConnection(url, username, password)
```

# MySQL - Security

- **Can use Docker to set an environment variable containing your DB password**

  - **Do not add the password when checking it into the repo**

- **When you're ready to deploy the app**

  - **Clone the repo, choose a password, and edit the files on the production server only**

  - **Access to this password should be on a need-to-know basis**

- **Alternatively/Additionally: Change your DB settings to only allow connections from localhost (Unless your app is distributed)**

```
val url = "jdbc:mysql://localhost/mysql?serverTimezone=UTC"
val username = "root"
val password = "12345678"

var connection: Connection = DriverManager.getConnection(url, username, password)
```

# MySQL

- Once connected, we can send SQL statements to the server

```
val statement = connection.createStatement()
statement.execute("CREATE TABLE IF NOT EXISTS players (username TEXT, points INT)")
```

- If using inputs from the user always use prepared statements

  - Indices start at 1 in this example 😢

```
val statement = connection.prepareStatement("INSERT INTO players VALUE (?, ?)")

statement.setString(1, "mario")
statement.setInt(2, 10)

statement.execute()
```

# MySQL - Security

- Not using prepared statements?

  - **Vulnerable to SQL injection attacks**

- If you concatenate user inputs directly into your SQL statements

  - Attacker chooses a username of "';DROP TABLE players;"

  - You lose all your data

  - Even worse, they find a way to access the entire database and steal other users' data

  - SQL Injection is the most common successful attack on servers

# MySQL

- Send queries to pull data from the database

- Returns a ResultSet in this example

  - The next() methods queue the next result of the query

  - next returns false if there are no more results to read

- Can read values by index of by column name

```
val statement = connection.createStatement()
val result: ResultSet = statement.executeQuery("SELECT * FROM players")

var allScores: Map[String, Int] = Map()

while (result.next()) {
  val username = result.getString("username")
  val score = result.getInt("points")
  allScores = allScores + (username -> score)
}
```

# SQL

- SQL is based on tables with rows and column

  - Similar in structure to CSV except the values have types other than string

- How do we store an array or key-value store?

  - With CSV our answer was to move on to JSON

  - SQL answer is to create a separate table and use JOINs

  - Or, try MongoDB

# MongoDB

- Runs on port 27017 (By default)

- A document-based database

- Instead of using tables, stores data in a structure very similar to JSON

- In python/JS

  - Insert dictionaries/objects directly

- Each object is stored in a collection

```
chat_collection.insert_one({'username': 'hartloff', 'message': 'hello'})
```

# MongoDB

- Retrieve documents using find

- Find takes a key-value store and returns all documents with those values stored at the given keys

  - Ex. {'username': 'hartloff'} returns all documents with a username of hartloff

- To retrieve all documents, use an empty key-value store {}

```
collection.find({'username': 'hartloff'})
collection.find({})
```

# MongoDB vs. SQL

- MongoDB is unstructured
  - Can add objects in any format to a collection
  - Can mix formats in a single collection
    - Ie. In a single collection the documents can have different attributes
- SQL is structured (That's what the S stands for)
  - Table columns must be pre-defined
    - All rows have the same attributes
    - Adding a column can be difficult
  - Fast!

# MongoDB vs. SQL

- Hot Take
  - MongoDB is best for prototyping when the structure of your data is constantly changing
    - Take advantage of the flexibility
  - SQL is best once your data has a defined structure
    - Take advantage of the efficiency