Databases

Project

- Change your passwords!!
  - See the pinned Piazza post and follow the instructions

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

```
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

```
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

```
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

Docker Compose

Docker Compose

- We need the application and database both running
  - App and database are 2 separate processes
- We'll use docker-compose to do this
- Let's walk through a docker-compose.yml file

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- Specify the docker compose file format version

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- List all of the services for docker compose to run
- A docker container is created for each service

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- Name each service
- These names are used as the hostnames for each container
  - Used to communicate between containers

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- This service named 'mongo' uses a pre-build image
  - Same as having a 1-line Dockerfile:
    - "FROM mongo:4.2.5"
- No Dockerfile is needed

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- This service named 'app' uses a Dockerfile
- Use 'build' to specify the path to build from
- Same as the trailing '.' when building an image

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- Use 'environment' to set any needed environment variables
- If using MySQL, set variables for your username/ password

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- We use an environment variable to tell our app to wait until the database is running before connecting to it

```
FROM python:3.8.2

ENV HOME /root
WORKDIR /root

COPY . .
RUN pip install -r requirements.txt

EXPOSE 8000

ADD https://github.com/ufoscout/docker-compose-wait/releases/download/2.2.1/wait /wait
RUN chmod +x /wait

CMD /wait && python app.py
```

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- If the app runs before the database, it won't be able to establish a DB connection
- Solution: Wait for the DB to start before running the app

```dockerfile
FROM python:3.8.2

ENV HOME /root
WORKDIR /root

COPY . .
RUN pip install -r requirements.txt

EXPOSE 8000

ADD https://github.com/ufoscout/docker-compose-wait/releases/download/2.2.1/wait /wait
RUN chmod +x /wait

CMD /wait && python app.py
```

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- This solution from github user "ufoscout" works well

```
FROM python:3.8.2

ENV HOME /root
WORKDIR /root

COPY . .
RUN pip install -r requirements.txt

EXPOSE 8000

ADD https://github.com/ufoscout/docker-compose-wait/releases/download/2.2.1/wait /wait
RUN chmod +x /wait

CMD /wait && python app.py
```

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- Map a local port to a container port
- Same as using "--publish 8080:8000" when running a single container

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

- This file is used to build both images and run both containers using docker-compose

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

~~`mongo_client = MongoClient('localhost')`~~

```
mongo_client = MongoClient('mongo')
```

- Recall that we chose names for each service
- When connecting to the database in your app
- The service name is the hostname for the container

Docker Compose

docker-compose.yml

```yaml
version: '3.3'
services:
  mongo:
    image: mongo:4.2.5
  app:
    build: .
    environment:
      WAIT_HOSTS: mongo:27017
    ports:
      - '8080:8000'
```

~~`mongo_client = MongoClient('localhost')`~~

```
mongo_client = MongoClient('mongo')
```

- Instead of using "localhost"/"127.0.0.1"/"0.0.0.0"
- Use the name of the service
- docker-compose will resolve this hostname to the appropriate container

Running Your App

- To run your app [and database]
  - docker-compose up
- To run in detached mode
  - docker-compose up -d
- To rebuild the containers
  - docker-compose build
  - docker-compose up --build