# 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 databases

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

- MongoDB

  - An unstructured server based on document stores

- mySQL

  - A server implementing SQL (Structured Query Language)

# MongoDB

- Runs on port 27017 (By default)

- A document-based database

- Stores data in a structure very similar to JSON

- In python/JS

  - Insert dictionaries/objects directly

- Each object is stored in a collection

# MongoDB - Connection

- Download a connection library and use to establish a connection with MongoDB

- MongoDB is separated into several layers

  - Databases - Named by Strings; Contains collections

  - Collections - Where the data is stored; similar to a SQL table

- Access your collections to insert/retrieve/update/delete data

```python
from pymongo import MongoClient

mongo_client = MongoClient("localhost")
db = mongo_client["cse312"]
chat_collection = db["chat"]
```

# MongoDB - Insert Data

- Insert dictionaries/objects directly

- For languages without a data structure comparable to dictionaries/objects

  - More work to do to prepare your data for Mongo

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

# MongoDB - Security

- No Mongo injection attacks

- Mongo does not rely on parsing statements as strings

- Any injected code would be treated as values

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

# MongoDB - Retrieve Data

- 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 {}

```
my_data = chat_collection.find({"username": "hartloff"})
all_data = chat_collection.find({})
```

# MySQL

- Listens for TCP connections on port 3306 (By default)

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

- SQL is based on tables with rows and column

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

# MySQL - Connection

- MySQL runs 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 - Insert Data

- 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 - Retrieve Data

- Send queries to pull data from the database

```scala
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)
}
```

# 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 Revisited

docker-compose.yml

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

- Let's modify our docker compose configuration to run our database

# Docker Compose

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

- "services" is a list of all the images/ containers to create

- We'll add a second service for the DB

# 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-built 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'
```

- 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

```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'
```

- 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

```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 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'
```

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

```python
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'
```

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


mongo_client = MongoClient('mongo')
```

- Use the name of the service

- docker-compose will resolve this hostname to the appropriate container

# Docker Compose

**docker-compose.yml**

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

```
mongo_client = MongoClient('localhost')


mongo_client = MongoClient('mysupercooldatabase')
```

- We can name our services whatever we want

- Make sure you are consistent!

# Running Your App

- docker-compose up --build --force-recreate

  - Will now start both containers

  - Use the service name as the host name to communicate across containers

# HTML Templates

# The Problem

- We want to serve custom HTML

- You want to build a chat feature for your app

  - Users will submit their messages

  - Messages will appear to all users

  - Messages are contained in your HTML

  - How do we serve HTML that will change as users send messages?

# HTML Templates

- Instead of writing complete HTML files

  - Write HTML templates

- An HTML template is an "incomplete" HTML file that is used to generate complete pages

- Use additional markup to add placeholders in the HTML

- Replace the placeholders with data at runtime

# HTML Templates

- Example template with 3 placeholders
- The title, description, and image_filename will be replaced later
  - Provide values for these 3 placeholders to serve a response

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

<h1>{{title}}</h1>

<p>{{description}}</p>

<img src="{{image_filename}}"/>

</body>
</html>
```

# HTML Templates

- To substitute the placeholders
  - Use any string manipulation that gets the job done
  - Find/replace is the simplest solution
  - May want more advanced approaches if you want to add more functionality

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

<h1>{{title}}</h1>

<p>{{description}}</p>

<img src="{{image_filename}}"/>

</body>
</html>
```

# Common Template Features

- Loops

- To add loops to your templates

  - Choose syntax for the start and end of the loop

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

{{loop}}
<h6>{{content_from_data_structure}}</h6>
{{end_loop}}

</body>
</html>
```

# Common Template Features

- Use string manipulation to find the start and end tags

- Iterate over your data

  - Add the contained HTML with the placeholder replaced for each value of your data

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

{{loop}}
<h6>{{content_from_data_structure}}</h6>
{{end_loop}}

</body>
</html>
```

# Common Template Features

- Conditionals

- Can use similar approach as loops

- Choose syntax for the start and end of each block in the conditional

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

{{if cookie_set}}
<h6>Welcome back!</h6>
{{else}}
<h6>Welcome!</h6>
{{end_if}}

</body>
</html>
```

# Common Template Features

- Search for your tags

- Extract and evaluate the conditional

  - Choose how this will be evaluated

- Add the appropriate block of HTML to the page

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This page was generated from a template</title>
</head>
<body>

{{if cookie_set}}
<h6>Welcome back!</h6>
{{else}}
<h6>Welcome!</h6>
{{end_if}}

</body>
</html>
```