Introduzione alle Applicazioni Web

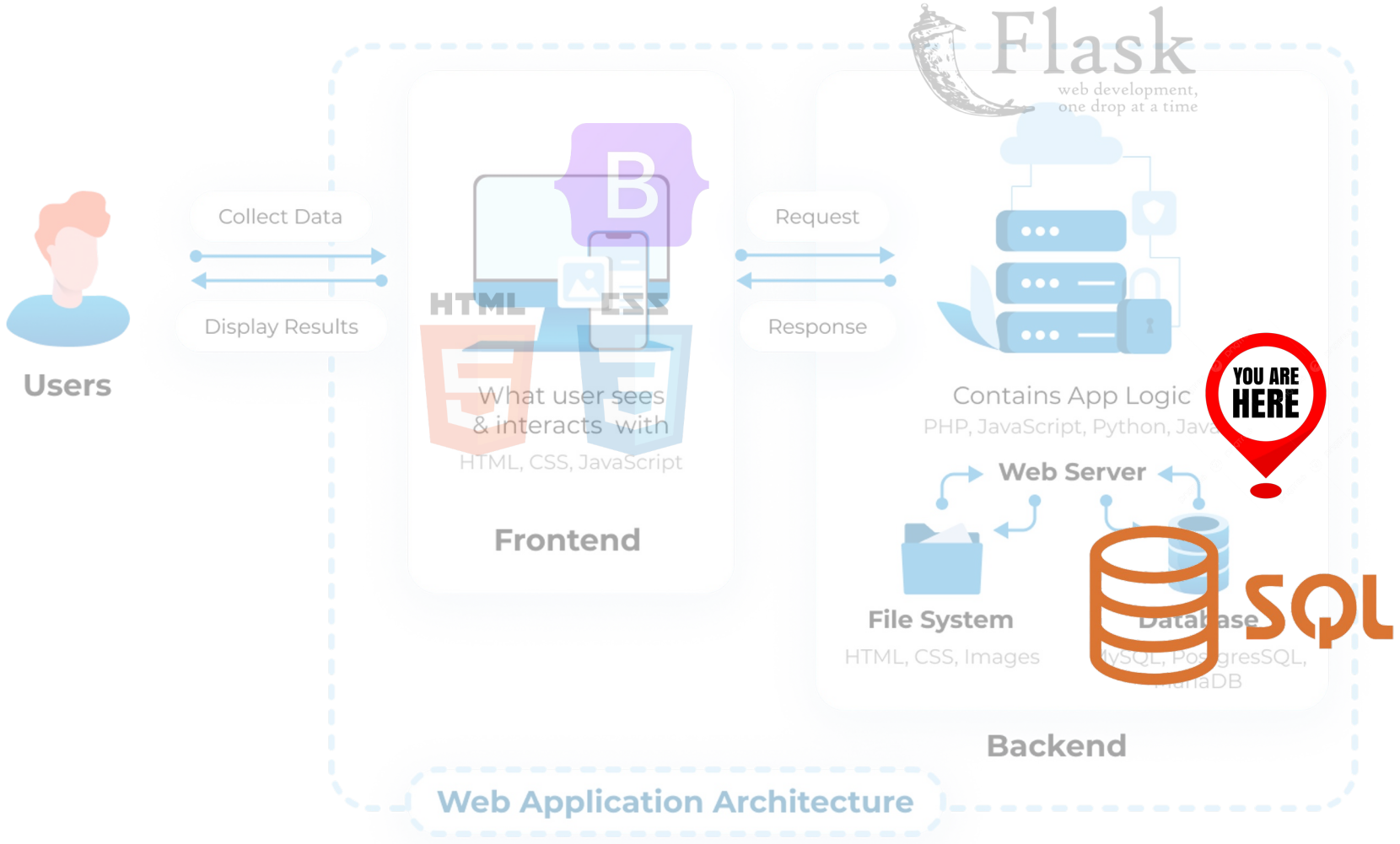
Database

Juan Pablo Sáenz

Goals

- Understand how to make **data persistent** across application restarts
- Learn how to manage large datasets **beyond in-memory storage**
- Use **SQL** to work with complex data and **perform advanced queries**

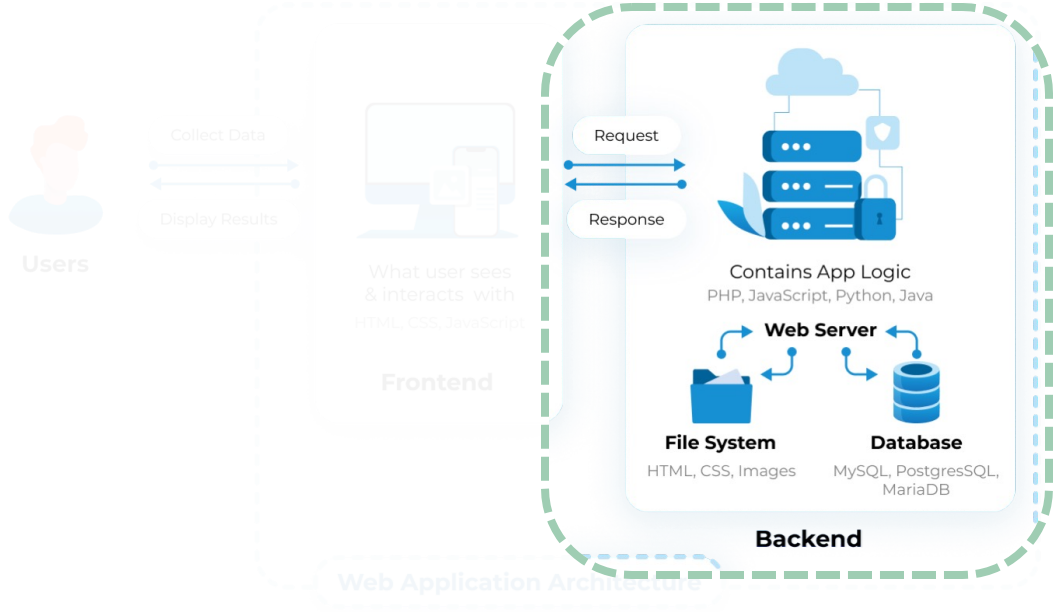
📍 Database: where are we?



Web architecture components: Backend

Database: a system that **stores and organizes data**, making it easy to retrieve, manage, and update.

- It ensures data integrity, security, and performance, often structured in **tables**, **rows**, and **columns**.
- **SQL (Structured Query Language)** is the language used to **interact with databases**, allowing users to search, insert, update, and delete data.

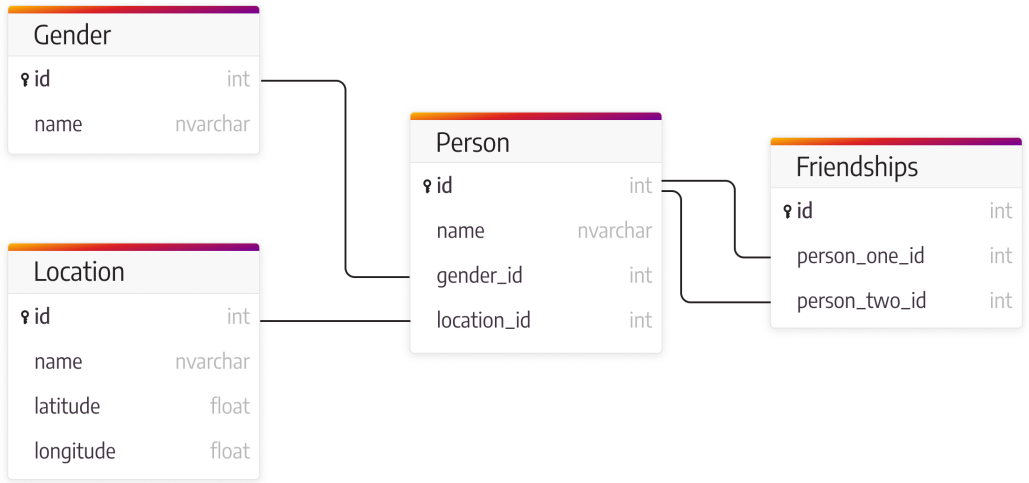


Relational databases

A **structured** way to store data in **tables** (**rows** and **columns**)

Each **table** represents a specific **entity** (e.g., users, products)

- ⚠️ Tables are named in **PLURAL** (e.g., users, orders) to reflect collections

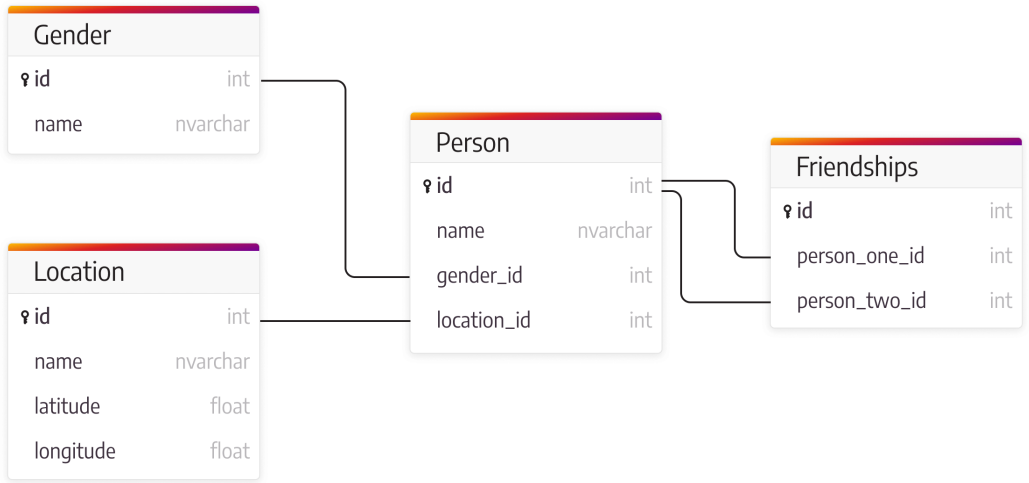


https://memgraph.com/blog/graph-database-vs-relational-database

Relational databases

Tables can be linked through **relationships** (e.g., foreign keys)

Enables **powerful querying using SQL** (Structured Query Language)

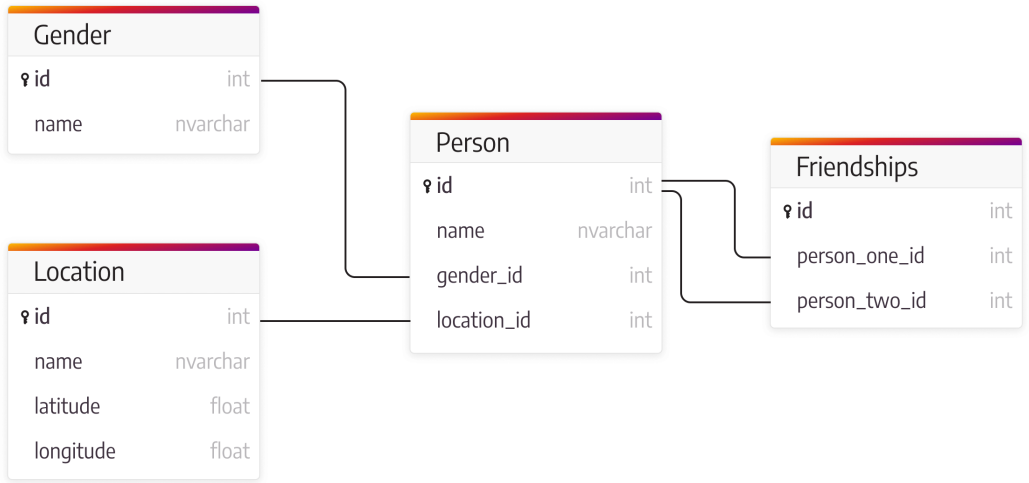


https://memgraph.com/blog/graph-database-vs-relational-database

Relational databases

Ensures data integrity through **ACID** properties:

- **A**tomicity: All or nothing: a transaction must **fully happen or not at all**
- **C**onsistency: **Data must stay correct** before and after a transaction
- **I**solation: **Transactions don't mess with each other**, even if run at the same time
- **D**urability: Once saved, **data won't be lost**—even if the system crashes



https://memgraph.com/blog/graph-database-vs-relational-database

Non-relational databases

Also known as **NoSQL** databases

Store data in **flexible formats** like:

- Documents (e.g., MongoDB)
- Key-value pairs (e.g., Redis)
- Graphs (e.g., Neo4j)

Do **not require a fixed schema**, making them ideal for **unstructured** or **evolving** data

Common in real-time applications

⚠ **Out of scope** for this course

Difference between Relational and Non-Relational databases

Relational

student

id	name	surname	age
1	John	Brown	19
2	Emma	Carly	23

Non-Relational

student.json file body:

```
[
 {
  "id": 1,
  "name": "John",
  "surname": "Brown",
  "age": 19
 },
 {
  "id": 2,
  "name": "Emma",
  "surname": "Carly",
  "age": 23
 }
]
```

https://codefinity.com/courses/v2/5ac24d9d-4a16-45b3-8856-07dec028c5e9/c9717913-3062-46d6-b0ef-164aca862b29/64995190-c55f-4482-a71f-ec51d6ae5b80

Relational databases: SQLite

A **lightweight**, **serverless** relational database

Stores data in a **single file** on disk

Commonly used in **mobile applications**, embedded systems, and for **rapid prototyping**

🤩 **Zero Configuration**: No separate server or complex setup required

Step-by-Step Guide to Using SQLite

Step 1: Importing SQLite

Since Python 2.5, **SQLite** has been included by default via the **`sqlite3`** module

```
# app.py

import sqlite3
```

Step-by-Step Guide to Using SQLite

Step 2: Query definition

- Use a regular SQL query stored in a **string**
- If your query includes variable parameters, use **placeholders** (**?**)
- ⚠️ **NEVER** build SQL queries by joining strings
- 📌 Use SQL templates with **placeholders** instead of inserting values directly
- 📌 Pass the actual values using **`.execute()`**

```
sql = "SELECT id, original,
modified FROM translation»

sql = "INSERT INTO translation
(original, modified) VALUES (?, ?)"
```

Step-by-Step Guide to Using SQLite

Step 2: Query definition

- Use a regular SQL query stored in a **string**
- If your query includes variable parameters, use **placeholders** (**?**)
- ⚠️ **NEVER** build SQL queries by joining strings
- 📌 Use SQL templates with **placeholders** instead of inserting values directly
- 📌 Pass the actual values using **`.execute()`**

```
sql = "SELECT id, original,
modified FROM translation"

sql = "INSERT INTO translation
(original, modified) VALUES (?, ?)»

# Define the query with a
placeholder
sql = "SELECT id, email FROM users
WHERE name = ?"
```

Step-by-Step Guide to Using SQLite

Step 3: Database connection

- Establishes a connection to a **SQLite database file** named **`example.db`**
- If the file does not exist, **SQLite will automatically create it**
- **`conn`** is a connection object used to:

  Create a **cursor** for executing SQL commands

  Commit changes

  Close the connection

```
conn =
sqlite3.connect('example.db')
```

Step-by-Step Guide to Using SQLite

Step 4: Query execution

- Get a **cursor** from the connection
- Execute a SQL query: **`.execute()`**
- Use **placeholders** to pass parameters safely
- Query parameters are given as a '**tuple**' argument

  ⚠️ One-element tuples require trailing `,`

  ```
  cursor.execute(sql, (txtid,))
  ```

```
sql = "INSERT INTO translation
(original, modified) VALUES (?, ?)"
# Create a cursor object to execute
SQL commands
cursor = conn.cursor()

# Insert a row using placeholders
(note the ? symbols)
cursor.execute(sql, ("Hello",
"Hola"))
```

Step-by-Step Guide to Using SQLite

Step 4: Query execution

- If the query was a **SELECT**

  `cursor.fetchone()`: retrieves the next result

  `cursor.fetchall()`: retrieves all remaining results

Both methods return tuples, representing the selected columns

https://www.python.org/dev/peps/pep-0249/#cursor-methods

```
# Fetch and print the results
rows = cursor.fetchall()
for row in rows:
    print(row)
```

Step-by-Step Guide to Using SQLite

Step 4: Query execution

- For **INSERT**, **UPDATE**, or **DELETE** there is no result
- The changes are not immediately applied to the database; they need to be '**committed**' first
- ⚠️ **Don't forget to commit**, or you might lose your data!
- This must be called before **`conn.close()`**

```
# Save (commit) the changes
conn.commit()

# Always close the connection when done
conn.close()
```

Step-by-Step Guide to Using SQLite

Step 5: Closing the cursor and the connection

- When you're done with the cursor, call **`cursor.close()`**
- Also, close the connection, this frees up resources on the database server: **`conn.close()`**

```
# Close the cursor
cursor.close()

# Always close the connection when done
conn.close()
```

Minimal example

```
conn = sqlite3.connect('example.db’)

sql_insert = "INSERT INTO translation (original, modified) VALUES (?, ?)"

cursor = conn.cursor()

cursor.execute(sql_insert, ("Hello", "Hola"))

conn.commit()

cursor.close()
conn.close()
```

Licenza

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/