# IERG 4210 Tutorial 3
# Phase 2B Server-Side Programming

2024 Spring

TA: CHEN Lin

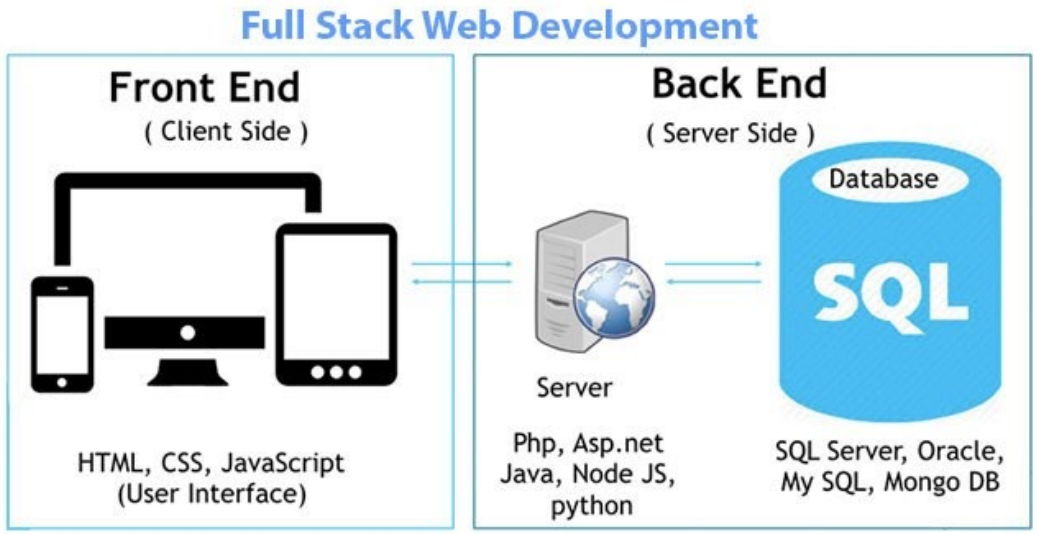
(Modified from the previous version by **XIE Siyue**)

# Review Phase 2A

- Amazon Web Service and EC2 Instance
- Environment Setup
  - Linux, Nginx, NodeJS and MySQL (or SQLite)
- Security Configurations

# Outline

- **Database and Basic SQL**
- Using SQLite in Node.js
- Server-side Request Handling



# Database

| Student ID(key) | Name | Age | Group |
|---|---|---|---|
| 115501 | A | 22 | 1 |
| 115502 | B | 23 | 2 |

- Relational Database
  - Tables (relations) of rows (records) and columns (attributes).
  - Each row has a unique key (one or many columns).

- RDBMS (Relational Database Management System)
  - MySQL
  - SQLite (We will use this one in this tutorial)
  - SQL Server
  - …

  (you are free to choose anyone, even the non-relational DBMS)

Categories

| CATID | NAME |
|---|---|
| 1 | Phone |
| 2 | Tablet |
| 3 | Laptop |

Each row is a record

Products

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |
| 2 | 1 | iPhone 13 pro | 8499 |
| 3 | 2 | iPad Air | 4799 |
| 4 | 2 | iPad Pro | 6399 |
| 5 | 3 | MacBook Air | 7799 |
| 6 | 3 | MacBook Pro | 9999 |

Each column is an attribute

Categories

Primary key: unique for each record in a table

| CATID | NAME |
|---|---|
| 1 | Phone |
| 2 | Tablet |
| 3 | Laptop |

Products

Primary key

Foreign key

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |
| 2 | 1 | iPhone 13 pro | 8499 |
| 3 | 2 | iPad Air | 4799 |
| 4 | 2 | iPad Pro | 6399 |
| 5 | 3 | MacBook Air | 7799 |
| 6 | 3 | MacBook Pro | 9999 |

# SQLite


- Public domain license (i.e., FREE!)
- Lightweight in design
  - Lightweight: multiple processes can read at the same time; however, only one process can make changes at any moment in time
  - Best for apps (MobileApps/Simple WebApps)
- Supported by multi-platforms (e.g., Windows, Linux)
- Stores everything in a single file
  - Easy to embed, test, backup and transfer
- Simple access-right management
  - No user account management as in full-blown DBs like MySQL
  - Depends on the file access rights

# SQLite

- Security Warning: Being the most important asset of any websites, SQLite DB file MUST NOT be accessible by the public web (end user); otherwise, it may raise serious security problems.
- Creating an SQLite database
  - Assume that `/ierg/pages/` is where you host your public pages (i.e., NEVER put your DB under the public directory `/ierg/pages/`)
  - To create a database at `/ierg named cart.db`
    - The required command, assuming AWS:
      `sudo sqlite3 /ierg/cart.db`
    - Enable foreign-key support
      sqlite> `PRAGMA foreign_keys = ON;`
    - SQLite command is case-insensitive
- Quit the SQLite command shell: `.quit`

More relevant SQLite commands: https://sqlite.org/cli.html

# SQLite: Create a Table

Categories

Primary key: unique for each record in a table

| CATID | NAME |
|---|---|
| 1 | Phone |
| 2 | Tablet |
| 3 | Laptop |

- Create a Table “categories”
  - Open the database if it exists
    ```
    sudo sqlite3 /ierg/cart.db
    ```
  - Create the table
    ```
    sqlite> CREATE TABLE categories (
              catid INTEGER PRIMARY KEY,
              name TEXT
            );
    ```
    Note: Primary key is unique and auto-increment by default

    (i.e., increase by 1 automatically for every new record)
  - To check what you have created:
    ```
    sqlite> .schema
    ```
- SQL Data Types: TEXT, INTEGER, REAL, …

More information on data types of SQLite: https://www.sqlite.org/datatype3.html

# SQLite: Create a Table (2/2)

- Creating a Table “products”
  - Create the table

```
sqlite> CREATE TABLE products (
          pid INTEGER PRIMARY KEY,
          catid INTEGER,
          name TEXT,
          price REAL,
          FOREIGN KEY(catid) REFERENCES categories(catid)
        );
```

Primary key

Foreign key

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |
| 2 | 1 | iPhone 13 pro | 8499 |
| 3 | 2 | iPad Air | 4799 |
| 4 | 2 | iPad Pro | 6399 |
| 5 | 3 | MacBook Air | 7799 |
| 6 | 3 | MacBook Pro | 9999 |

  - Create an index for catid (to make subsequent queries by catid faster)

```
sqlite> CREATE INDEX i1 ON products(catid);
```

  - Delete the table in case you did something wrong

```
sqlite> DROP TABLE products;
```

More information on creating index: https://sqlite.org/lang_createindex.html

# SQLite: INSERT

- Insert some records into the table
  - Insert a record into “categories”

    sqlite> `INSERT INTO categories VALUES (NULL, "Phone");`

    Note: “null” for auto-increment of the value of primary key
  - Insert a records into “products”

    sqlite> `INSERT INTO products`
    `VALUES (NULL, 1, "iPhone 13", 6799););`

    sqlite> `INSERT INTO products (catid, name, price)`
    `VALUES (1, "iPhone 13", 6799);`

    Note: If the value of `CATID` does not exist in `Categories` , it will raise an error.

Categories

| CATID | NAME |
|---|---|
| 1 | Phone |

Products

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |

More on SQL INSERT: http://www.sqlite.org/lang_insert.html

# Constraints

- Constraints can restrict the query
  - Specify rules and abort some records that violate the rules
  - NOT NULL, UNIQUE , PRIMARY KEY , FOREIGN KEY , CHECK , DEFAULT
- Primary Key
  - With unique value to distinguish each record
  - A combination of NOT NULL and UNIQUE

```
sqlite> CREATE TABLE products (
   pid INTEGER PRIMARY KEY,
   catid INTEGER,
   name TEXT,
   price REAL,
   FOREIGN KEY(catid) REFERENCES categories(catid) );
```

```
sqlite> PRAGMA foreign_keys = ON;
sqlite> INSERT INTO products (catid,name,price)
VALUES (2,"Help","88");
Error: foreign key constraint failed
sqlite>
```

- Foreign Key
  - Bind one row with another row in a different table
  - If you try to insert a product to an inexistent category:

```
sqlite> INSERT INTO products (catid, name, price)
        VALUES (2, "Help", "88");
Error: constraint failed
```

(expected error given that the foreign key feature is ON)

# SQLite: Query

Products

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |
| 2 | 1 | iPhone 13 pro | 8499 |

- Looking up records
  - To output all the records in the products table:

    sqlite> `SELECT * FROM products`
  - To select all “phone” in products (given ”phone” is of catid=1):

    sqlite> `SELECT * FROM products WHERE catid = 1;`
  - To select only the name and price columns

    sqlite> `SELECT name, price FROM products WHERE catid = 1;`
  - To select phones with a price no more than 8000

    sqlite> `SELECT * FROM products WHERE catid = 1 AND price <= 8000;`

More on SQL SELECT: http://www.sqlite.org/lang_select.html

# SQLite: UPDATE

Products

| PID | CATID | NAME | PRICE |
|---|---|---|---|
| 1 | 1 | iPhone 13 | 6799 |
| 2 | 1 | iPhone 13 pro | 8499 |

- Update a record
  - Update a value (update the price into 5999 of the product with PID = 1)

```
sqlite> UPDATE products
        SET price = 5999
        WHERE pid = 1;
```

  - Update through an expression (decrease the price for all phone products by 1000)

```
sqlite> UPDATE products
        SET price = price - 1000
        WHERE catid = 1;
```

- Note:
  - The usage of WHERE is the same as that of SELECT
  - So, when you are not sure about what records will be affected
    - SELECT the records first (and check), then replace SELECT with UPDATE
    - Otherwise, you may kill all your data unintentionally

More on SQL UPDATE: http://www.sqlite.org/lang_update.html

# SQLite: DELETE

- Delete a record
  - Delete a product by pid
    sqlite> `DELETE FROM products WHERE pid = 2;`
  - Delete the category will raise an error
    sqlite> `DELETE FROM categories WHERE catid = 1;`
    Reason: Given foreign key is ON, a record with children can't be deleted

In this case, the record `(1, "Phone")` in `Categories` still has a child `(1, 1, "iPhone 13", 6799)` in `Products` , thus it cannot be deleted. If you want to delete the record in `Categories` , delete all its children in `Products` first.

- Be attention when deleting records
  - Backup your database
  - Or SELECT affected rows before performing DELETE

More on SQL DELETE: http://www.sqlite.org/lang_delete.html

# Outline

- Database and Basic SQL
- **Using SQLite in Node.js**
- Server-side Request Handling

# Using SQLite3 in Node.js

- Adding the Sqlite3 package
  open your terminal in the desired folder and simply type

```
cd ~/ierg                  # Setting Up the Project Directory
npm install sqlite3        # set up the project with the package
touch ~/ierg/test.js       # create test.js
touch ~/ierg/test.db       # create test.db
```

- Connecting to an SQLite3 database
  - Edit test.js

```
nano ~/ierg/test.js
```

  - Paste the following content to `test.js`

```
const sqlite3 = require("sqlite3").verbose();


//connect to DB
const db = new sqlite3.Database("./test.db", sqlite3.OPEN_READWRITE, (err) => {
  if (err) return console.error(err.message);
});
```

  - Run `test.js` in terminal

```
node test.js
```

# Using SQLite3 in Node.js

- Creating your first table named users
  - Edit test.js

```
let sql;
sql = `CREATE TABLE users (id INTEGER PRIMARY KEY, first_name, last_name, username, password, email)`;
db.run(sql);
```

```
const sqlite3 = require("sqlite3").verbose();

let sql;

//connect to DB
const db = new sqlite3.Database("./test.db", sqlite3.OPEN_READWRITE, (err) => {
  if (err) return console.error(err.message);
});

//create table :
sql = `CREATE TABLE users (id INTEGER PRIMARY KEY, first_name, last_name, username, password, email)`;
db.run(sql);
```

- Run `test.js` in terminal

  `node test.js`

```
ubuntu@ip-172-31-87-112:~/ierg$ node test.js
ubuntu@ip-172-31-87-112:~/ierg$ node test.js
undefined:0


[Error: SQLITE_ERROR: table users already exist
Emitted 'error' event on Statement instance at:
] {
  errno: 1,
  code: 'SQLITE_ERROR'
}

Node.js v20.11.1
ubuntu@ip-172-31-87-112:~/ierg$
```

# Using SQLite3 in Node.js

- Inserting data

```
sql = `INSERT INTO users (first_name, last_name, username, password, email) VALUES(?,?,?,?,?)`;
db.run(
  sql,
  ["mike", "michaelson", "mike_user", "test", "mike@gmail.com"],
  (err) => {
    if (err) return console.error(err.message);
  }
);
```

- Querying data

```
sql = `SELECT * FROM users`;
db.all(sql, [], (err, rows) => {
  if (err) return console.error(err.message);
  rows.forEach((row) => {
    console.log(row);
  });
```

```
ubuntu@ip-172-31-87-112:~/ierg$ node test.js
{
  id: 1,
  first_name: 'mike',
  last_name: 'michaelson',
  username: 'mike_user',
  password: 'test',
  email: 'mike@gmail.com'
}
ubuntu@ip-172-31-87-112:~/ierg$
```

```
 GNU nano 6.2                                /home/ubuntu/ierg/test.js *
//drop table :
// db.run("DROP TABLE users");

//insert data into table :
//sql = `INSERT INTO users (first_name, last_name, username, password, email) VALUES(?,?,?,?,?)`;
//db.run(
//   sql,
//   ["mike", "michaelson", "mike_user", "test", "mike@gmail.com"],
//   (err) => {
//     if (err) return console.error(err.message);
//   }
// );

//querying the data :
sql = `SELECT * FROM users`;
db.all(sql, [], (err, rows) => {
  if (err) return console.error(err.message);
   rows.forEach((row) => {
    console.log(row);
   });
   });
```

# Using SQLite3 in Node.js

- Deletee data

```
sql = `DELETE FROM users WHERE id = ?`;
 db.run(sql, [1], (err) => {
  if (err) return console.error(err.message);
 });
```

- Update data

```
sql = `UPDATE users SET first_name = ? WHERE id = ?`;
 db.run(sql, ["Jake", 1], (err) => {
   if (err) return console.error(err.message);
 });
```

```
ubuntu@ip-172-31-87-112:~/ierg$ node test.js
{
  id: 1,
  first_name: 'Jake',
  last_name: 'michaelson',
  username: 'mike_user',
  password: 'test',
  email: 'mike@gmail.com'
}
```

```
//update data :
sql = `UPDATE users SET first_name = ? WHERE id = ?`;
db.run(sql, ["Jake", 1], (err) => {
   if (err) return console.error(err.message);
 });

//querying the data :
sql = `SELECT * FROM users`;
db.all(sql, [], (err, rows) => {
  if (err) return console.error(err.message);
   rows.forEach((row) => {
    console.log(row);
   });
   });
```

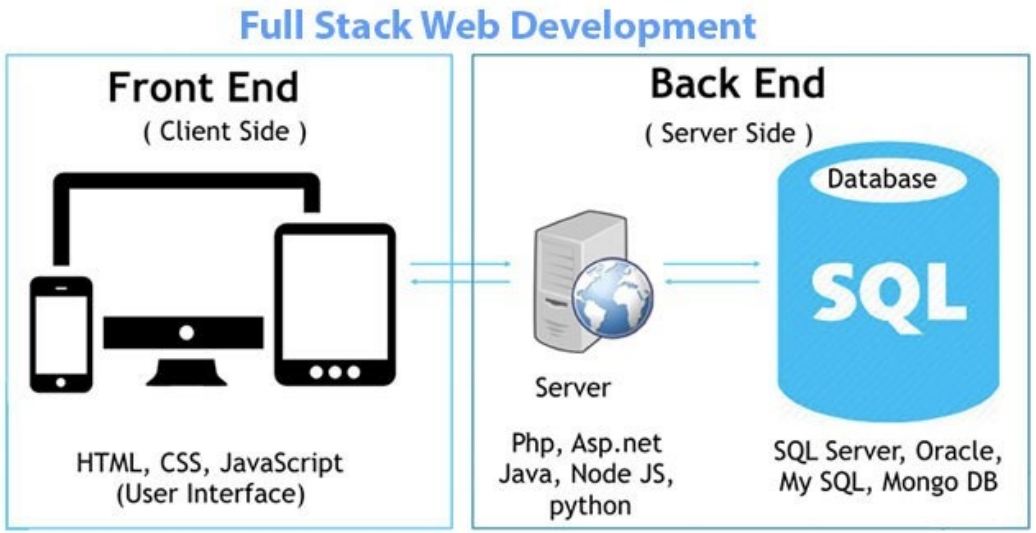
[How To Use SQLite with Node.js on Ubuntu 22.04]

# Outline

- Database and Basic SQL
- Using SQLite in Node.js
- **Server-side Request Handling**

# Web Development + Database Manipulation

- Workflow
  - User sends requests
  - Server handles requests
    - Query records from database
      - Or delete/ update/ etc.
    - Other preprocessing
    - Render HTML by populating preprocessed info
  - Display the webpage to the user

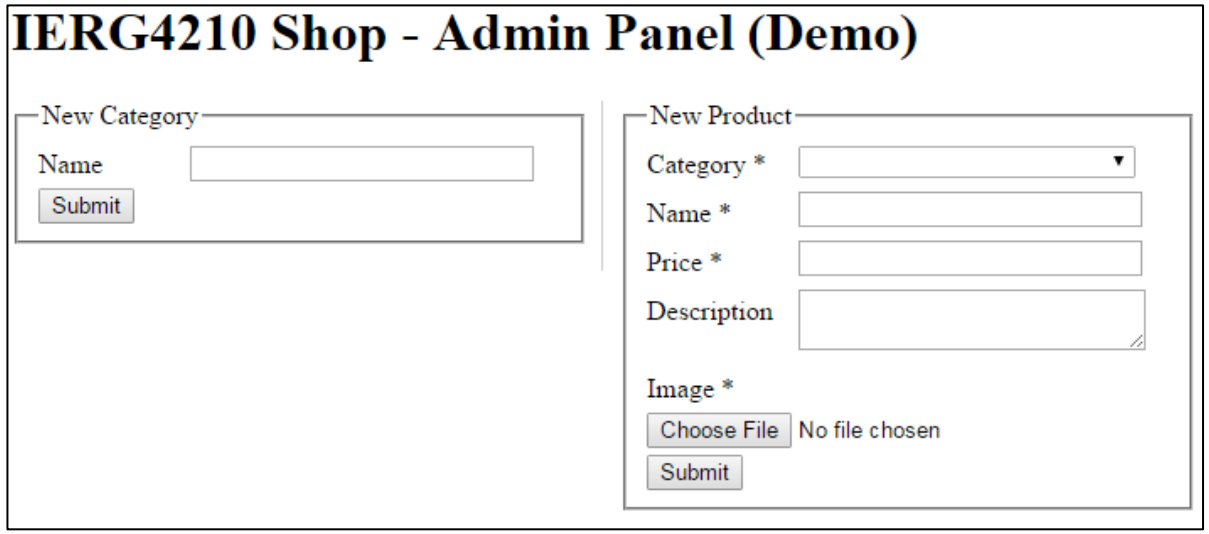


# Server-Side Request Handling

- “GET” request
  - requests a specific resource from server
  - parameters are appended in URL
  - Security issues
    - response can be cached, bookmarked; remained in browser history; etc.
- “POST” request
  - submits data to server
  - parameters are encoded and invisible in the URL
  - support file uploading

# Server-side Request Handling (e.g., HTML form)

- Admin Panel
  - Collect requests of adding /editing category / product info through HTML form
  - Server processes the requests and manipulates the database




**HTML form with GET and POST methods in Node.js**

Node.js Upload Files