Intro to Databases

Where are we going...

1. Web page structure and appearance with HTML5 and CSS.
2. Client-side interactivity with JS DOM and events.
3. Using web services (API's) as a client with JS.
4. **Writing JSON-based web services with a server-side language.**
5. Storing and retrieving information in a database with SQL and server-side programs.

Introduction to Databases

What is a Database?

A collection of related information, similar to JSON (but more tabular)

What are some examples of data you could store in a database?

- Pokedex
- Book Reviews
- Store Inventory
- User Information
- Movies/Games
- Cafe Menu
- ...

Why Learn Databases in this Class?

Databases give us a great improvement in the way we can build, process, and retrieve large datasets. Most software companies will have a large group dedicated to database management.

Advantages of a database:

- **Powerful**: Can search it, filter data, combine data from multiple sources.
- **Fast**: Can search/filter a database very quickly compared to a file/JSON.
- **Big**: Scale well up to very large data sizes.
- **Safe**: Built-in mechanisms for failure recovery (e.g. transactions).
- **Multi-user**: Concurrency feature let many users view/edit data at the same time.
- **Abstract**: Provides layer of abstraction between stored data and app(s) - many database programs understand the same commands.

Database Software

Oracle
Microsoft SQL Server (powerful) and Microsoft Access (simple)
PostgreSQL (powerful/complex free open-source database system)
SQLite (transportable, lightweight free open-source database system)
MySQL (simple free open-source database system)
Types of databases not covered in our course:

- NoSQL
- Key/Value
- Blob/Document
- GraphQL
- HBase, distributed databases

Relational Database vs. Others

Many different ways to store data (see previous slide)

Relational databases define tabular data (data stored in tables), with operations that allow you to relate them to each other.

Modern RDBMSs (**R**elational **D**ataBase **M**anagement **S**ystem) are transactional.

This allows the database to ensure that certain expectations about your data are met.

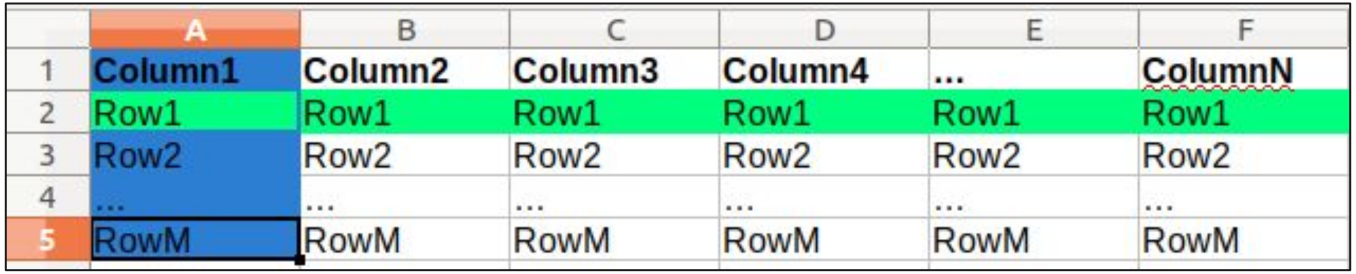
Each transaction is made up of 1+ commands

A Relational Database

Relational Database: A method of structuring data as tables associated by shared attributes

A table row corresponds to a unit of data called a **record**; a column corresponds to an attribute of that record

In Excel-speak:

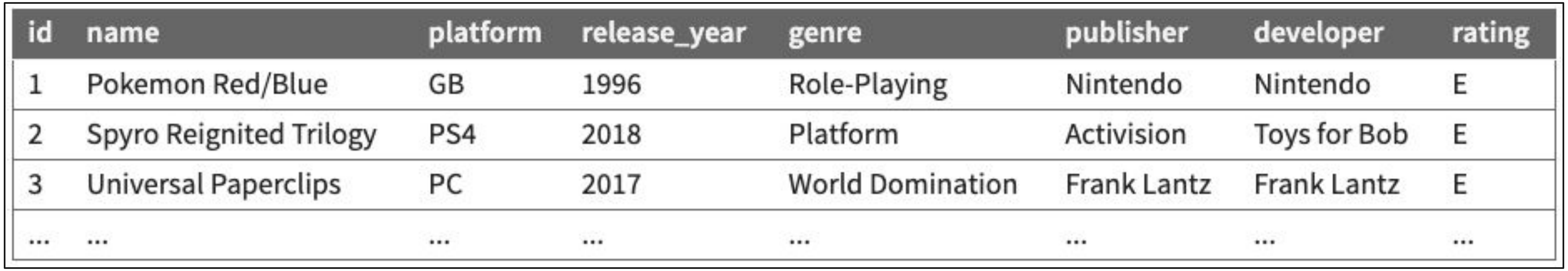


	A	B	C	D	E	F
1	**Column1**	**Column2**	**Column3**	**Column4**	**...**	**ColumnN**
2	Row1	Row1	Row1	Row1	Row1	Row1
3	Row2	Row2	Row2	Row2	Row2	Row2
4	...	...	...	...	...	...
5	RowM	RowM	RowM	RowM	RowM	RowM

In the above image, the cells highlighted blue are the first "column" and the cells highlighted green are of the first "row" (or "record", or "tuple")

Tables can be visualized just like an Excel sheet, just with different terminology, and more programmatic capabilities.

Example Database



id	name	platform	release_year	genre	publisher	developer	rating
1	Pokemon Red/Blue	GB	1996	Role-Playing	Nintendo	Nintendo	E
2	Spyro Reignited Trilogy	PS4	2018	Platform	Activision	Toys for Bob	E
3	Universal Paperclips	PC	2017	World Domination	Frank Lantz	Frank Lantz	E
...	...	...	...	...	...	...	...

Structured Query Language (SQL)

A "domain-specific language" (HTML is also a DSL) designed specifically for data access and management.

Also like HTML, SQL is a **declarative** language: describes what data you are seeking, not exactly how to find it.

In SQL, you write statements. The main different types of statements we'll look at:

- **Data Definition**: Generally, what does your data look like?
- **Data Manipulation**: Change or access the data.

(There are others, but we won't be talking about them.)

Today starts our intro to SQL

```
function filterJSON(pokemonJSON) {
  let pokemon = pokemonJSON["pokemon"];
  let filtered = [];
  for (let i = 0; i < pokemon.length; i++) {
    let data = pokemon[i];
    if (data.weakness === "rock" && data.name.indexOf("r") !== -1 && data.id < 145) {
      filtered.push({
        "name": data.name, "type": data.type,
        "id": data.id, "weakness": data.weakness
      });
    }
  }
}
```

The following code does everything above without JSON/JS!

```
SELECT name, id, type, weakness
FROM pokemon
WHERE id < 145 AND name LIKE '%a%' AND weakness = 'rock'
ORDER BY type, name DESC;
```

A Preview to Node.js + SQL

```
let qry = "SELECT name, id, type, weakness FROM pokedex " +
          "WHERE name LIKE '%r%' AND id < 145 AND weakness = 'rock' " +
          "ORDER BY type, name DESC";
let rows = await db.all(filterQuery);
console.log(JSON.stringify(rows));
// [{"name":"Articuno","id":144,"type":"ice","weakness":"rock"}]
```

Querying for rows using `.all`

We can use the `.all` function to get an <u>array</u> of rows returned by the passed query.

```
let db = await getDBConnection();
let ex1 = await db.all('SELECT * FROM cafe');
let ex2 = await db.all('SELECT * FROM cafe WHERE price = ?', '5');
```

Executing a statement using `.run`

The run function executes a single SQL query

```
let db = await getDbConnection();
let sql = 'INSERT INTO pokedex (name, type) VALUES (?, ?)';
await db.run(sql, ['Bulbasaur', 'grass']);
```

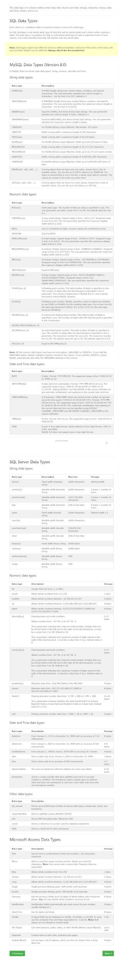
Types in SQLite

SQLite (sort of like JavaScript) has a lot of built-in flexibility.

It only *actually* has 5 data types:

- **NULL** -- like in JavaScript, stands for the absence of value
- **INTEGER** -- whole numbers (e.g., -3, -2, -1, 0, 1, 2, 3, 4, etc.)
- **REAL** -- decimal numbers (e.g., 3.141592653589)
- **TEXT** -- strings
- **BLOB** -- everything else

However: If you look up "SQL data types" on The Internet™ you'll find pages and page of different types



SQLite uses dynamic typing

Declaring variables in **JavaScript**: `var`, `let`, `const`. ← **Dynamic**

Declaring variables in **Java**: `int`, `String`, `double`, `Point`, etc. ← **Static**

SQLite uses dynamic typing

Declaring variables in **JavaScript**: `var`, `let`, `const`. ⟵ **Dynamic**

Declaring variables in **Java**: `int`, `String`, `double`, `Point`, etc. ⟵ **Static**

Declaring columns in **SQLite**: `INTEGER`, `TEXT`, etc. ⟵ **Dynamic**

Declaring columns in ***just about every other SQL (relational) DB:*** . . . . ⟵ **Static**

SQL Data Types

The following are commonly used *static* SQL types:

Type	Argument	Description
INT	none	An integer number
VARCHAR(n)	The string can be at most n characters (max of 65,535).	A text string
TINYINT	none	Stores only very small numbers (definition of "small" varies, but is usually between 0 and 255)
DATETIME	none	Stores dates and times for precise time information: YYYY-MM-DD HH:MM:SS
DOUBLE	Optionally (m,d), specifying how many digits in the number.	A decimal number. Rounds based on the provided precision.
TEXT	none	A potentially very large text string.

Conversion to SQLite

Static SQL Type	SQLite Type
INT	INTEGER
VARCHAR(n)	TEXT
TINYINT	INTEGER
DATETIME	NUMERIC
DOUBLE	REAL
TEXT	TEXT

Conversion to SQLite

Static SQL Type	SQLite Type
INT	INTEGER
VARCHAR(n)	TEXT
TINYINT	INTEGER
DATETIME	<u>**NUMERIC**</u>
DOUBLE	REAL
TEXT	TEXT

SQLite has this special value classification for things you might want to have handled specially.

These include:

- Dates
- Booleans
- "Strings"

CREATE TABLE: Syntax

WHAT IS THIS?!?!

```
CREATE TABLE table_name(
  column1 datatype PRIMARY KEY,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype
);
```

Useful Column Constraints

The following are very common and useful in CREATE TABLE statements.
These are called constraints - they "constrain" the types of values you can insert in a column.
This reading on constraints is an excellent overview for more details.

- PRIMARY KEY (keyname): Used to specify a column or group of columns uniquely identifies a row in a table.
- AUTOINCREMENT: Used with the primary key column to automatically generate the "next" value for the key. In SQLite, only available for the primary key.
- NOT NULL: prevents NULL entries in a column, requires the value to be set in INSERT statements.
- DEFAULT: specifies default values for a column if not provided in an INSERT statement
- UNIQUE: requires an attribute to be unique (useful for fields that are not PRIMARY KEY but should still be unique)

Extracting the data

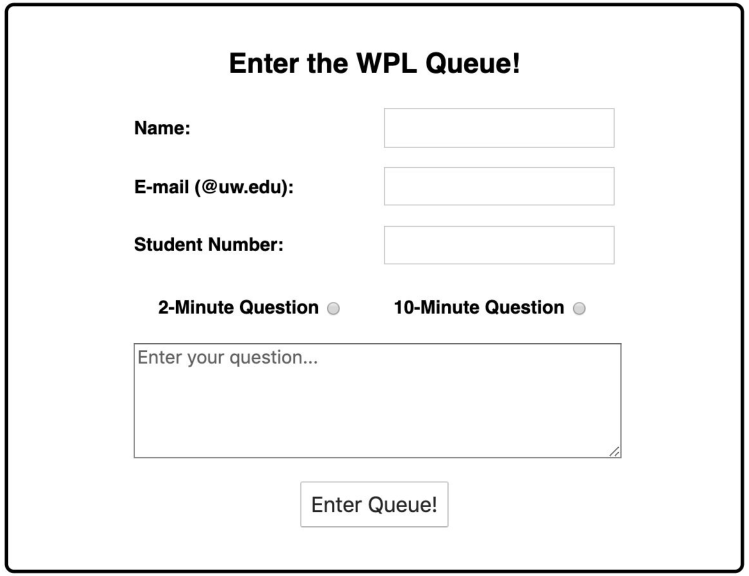
Every table should have a column which is used to uniquely identify each row. This improves efficiency and will prove very useful when using multiple tables.

```
CREATE TABLE students(
  id        INTEGER  PRIMARY KEY AUTOINCREMENT,
  name      TEXT     NOT NULL,
  username  TEXT     NOT NULL UNIQUE,
  email     TEXT     NOT NULL
);
```

Adding `PRIMARY KEY` makes it so that the code will error if that column ever has duplicates. It will use that column to identify each row quickly. This is usually an integer id, but can also be other types. Conventionally named `id`.
`AUTOINCREMENT` will make it so that if you provide no input for that column, it will pick the highest unused value. Perfect for making it so you don't have to worry about what the next id is.

Remember the WPL Queue?

We introduced the WPL queue to teach Forms and validation on the client-side. What types of data might we want to store in a database (`wpl.db`) for the WPL Queue?



Enter the WPL Queue!

Name:	
E-mail (@uw.edu):	
Student Number:	

2-Minute Question ○ **10-Minute Question** ○

Enter your question...

Enter Queue!

Text-based

- The name of the student
- The email address
- The question text

But what about...

- The student number (this could also be text)
- The question length (2 or 10)
- A unique identifier for each question
- When the question was submitted

CREATE TABLE: example

What would the `CREATE TABLE` SQL command look like to create a table to hold the *queue* for the WPL example?

```
CREATE TABLE queue(
   id              INTEGER PRIMARY KEY AUTOINCREMENT,
   name            TEXT,
   email           TEXT,
   student_id      INTEGER,
   length          INTEGER,
   question        TEXT,
   creation_time   DATETIME DEFAULT CURRENT_TIMESTAMP
);
```