Database Theory

Part I - Topics Covered

- Database Tables - What they are and how they are created
- Data Types - What kind of values can be stored in a column
- Constraints - Enforcing rules on what type of data can go where
- Relationships - How data from one table can reference data from another table

Why learn about databases?

- A website without a database is just a static collection of text
- Databases efficiently store and retrieve information
- This information often comes from HTML form submissions

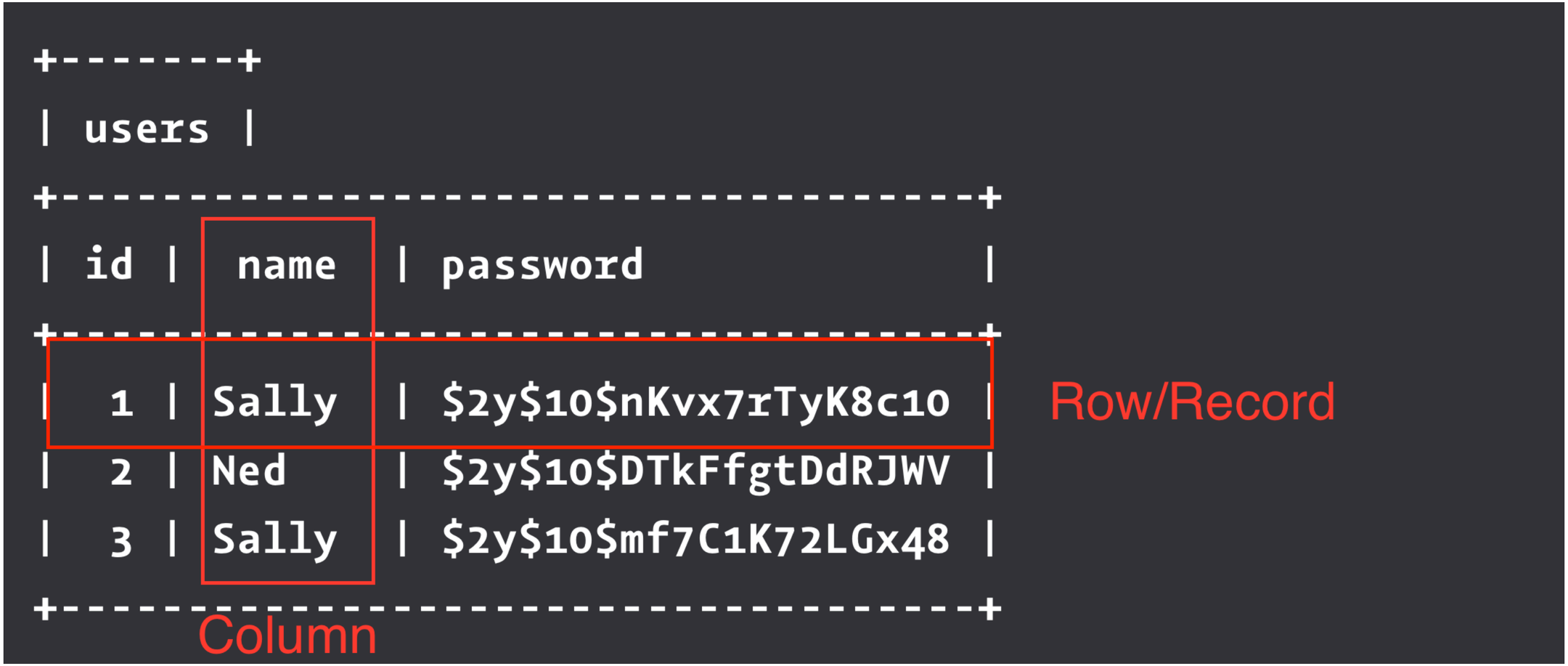
What is a database?

- Data can be inserted, read, updated, and destroyed from the database
- The database itself lives in a file on your computer
- Databases have one or more **tables**

Tables

- Every table stores some unique category of data
- A **"users" table** might store user information like name, email address, and password.
- A **"products" table** might store information like product name and price.

Tables



Tables

```
+-------+
| users |
+-----------------------------------+
| id |  name  | password             |
+-----------------------------------+
|  1 | Sally  | $2y$10$nKvx7rTyK8c10 |
|  2 | Ned    | $2y$10$DTkFfgtDdRJWV |
|  3 | Sally  | $2y$10$mf7C1K72LGx48 |
+-----------------------------------+
```

- The ID column is the unique identifier for the particular row it belongs to
- It is referred to as a **PRIMARY KEY**
- Almost every table will include an ID column for storing unique IDs
- The unique ID is an Auto Incrementing ID (automatically assigned)

Tables

```
+-------+
| users |
+-----------------------------------+
| id |  name  | password             |
+-----------------------------------+
|  1 | Sally  | $2y$10$nKvx7rTyK8c10 |
|  2 | Ned    | $2y$10$DTkFfgtDdRJWV |
|  3 | Sally  | $2y$10$mf7C1K72LGx48 |
+-----------------------------------+
```

- You don't need to manually assign the number value of the auto incrementing ID
- We might have more than one person named Sally but these are two different people
- Rows can contain similar data
- A unique ID is used to differentiate rows and create relations (more on relations later)

Tables

```
+-------+
| users |
+-----------------------------------+
| id |  name  | password             |
+-----------------------------------+
|  1 | Sally  | $2y$10$nKvx7rTyK8c10 |
|  2 | Ned    | $2y$10$DTkFfgtDdRJWV |
|  3 | Sally  | $2y$10$mf7C1K72LGx48 |
+-----------------------------------+
```

- The name column contains a simple string value
- The password column contains a "hashed" or encrypted version of the persons actual password

Tables

- The database schema provides its entire structure
- The structure includes **tables** and **columns**
- Every column has a **data type**

Data Types

When defining a table you must specify a data type for every column you add to it.
There are many data types and we will not cover all of them. **Some** common data types include:

- **VARCHAR**: Variable length string of characters
- **INTEGER**: Number with no decimal (5)
- **DECIMAL**: Number with decimal (50.10)
- **BOOLEAN**: Stores TRUE/FALSE values (1 or 0)
- **DATETIME**: Stores a date and time value

Please **write these down**. You will need them later.

Database Management Systems

- Tables are created through SQL (Structured Query Language)
- There are many forms of SQL, most of which are fairly similar
- Different forms of SQL are found in different DBMS's (Database Management Systems)

Database Management Systems

- MySQL, SQLite, PostgreSQL, Microsoft SQL Server are all popular DBMS's
- The different DBMS's are not very important right now
- Just know that the SQL syntax varies between the different DBMS's
- We will be using MySQL as an example

Basic SQL Table

```
CREATE TABLE people (
  username VARCHAR(255),
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  age INT
);
```

- Between the parenthesis the first value is the column name
- The second value is the **DATA TYPE**
- If the column should hold a number, you use the **INT** type
- If the column should hold a mixture of characters you use the **VARCHAR** type
- The VARCHAR type has a length parameter. We use 255 because it is suitable and common.
- The code between the parenthesis defines the **schema** for the table

Constraints

- PRIMARY KEY - Value(s) in specified column(s) must be unique for each row in a table and not be NULL
- UNIQUE - Value(s) in specified column(s) must be unique for each row in a table
- NOT NULL - Values for the column must not be NULL
- DEFAULT - Provide a default value for the column if no value is provided
- FOREIGN KEY - Value(s) in specified column(s) must reference an existing record in another table

Please **write these down**. You will need them for an exercise. We will discuss foreign key later.

SQL Table With Constraints

```
CREATE TABLE people (
  username VARCHAR(255) NOT NULL UNIQUE,
  first_name VARCHAR(255),
  last_name VARCHAR(255) NOT NULL,
  age INT DEFAULT 21
);
```

- When a column is NOT NULL then a row cannot be inserted that has no value for that column
- For example, we don't want any user record in our database with no username
- Lack of a username could cause bugs or broken features in our codebase so we ensure that one exists for every user
- The DEFAULT constraint provides a default value for a column value when no value is provided
- The UNIQUE constraint ensures that no two records can share the same value for the given column

Constraints - Review

- Correct data is very important to a stable program
- Constraints on the DB level enforce good data
- Make your DB as strict as possible
- If your application expects a value to be unique, you must remember to create that constraint
- If a value should never be empty, you must enforce the NOT NULL constraint

PRIMARY KEY

Most tables include an ID column. It holds a unique value for each row in a table.

Most ID columns are automatically populated or set to "auto increment".

```
CREATE TABLE people (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY
  username VARCHAR(255) NOT NULL UNIQUE,
  first_name VARCHAR(255),
  last_name VARCHAR(255) NOT NULL,
  age INT DEFAULT 21
);
```

Exercise - Create SQL Table Together

Lets create a users table with the following columns:

- `id`
- `name`
- `email`
- `password`
- `created_at`
- `updated_at`
- `num_logins`
- `is_email_address_confirmed`

Exercise - Create SQL Table Together

```
CREATE TABLE users (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL,
  created_at DATETIME NOT NULL,
  updated_at DATETIME NOT NULL,
  num_logins INT NOT NULL DEFAULT 0,
  is_email_address_confirmed TINYINT(1) NOT NULL /* boolean */
);
```

- TINYINT is typically used for boolean types in MySQL
- Data types can vary between DBMS. PostgreSQL has a data type actually called "boolean"
- We don't want any duplicate email addresses in our database so we enforce that with the UNIQUE constraint

Exercise - Create Products Table

- Pair up and help each other create a products table
- Include the columns `id`, `name`, `price`, `is_in_stock`, `sku`
- SKU is unique ID for product, can contain letters and numbers
- Assign the correct data type to each column
- Assign a PRIMARY KEY (auto incrementing)
- Assign a UNIQUE KEY to the appropriate column
- Reference the users table for tips

```
CREATE TABLE users (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL,
  created_at DATETIME NOT NULL,
  updated_at DATETIME NOT NULL,
  num_logins INT NOT NULL DEFAULT 0,
  is_email_address_confirmed TINYINT(1) NOT NULL /* boolean */
);
```

Exercise - Create Products Table - Answer

- Pair up and help each other create a products table
- Include the columns `id`, `name`, `price`, `is_in_stock`, `sku`
- SKU is unique ID for product, can contain letters and numbers
- Assign the correct data type to each column
- Assign a PRIMARY KEY (auto incrementing)
- Assign a UNIQUE KEY to the appropriate column
- Reference the users table for clues

```
CREATE TABLE products (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  price INT NOT NULL, /* Could also use DECIMAL */
  is_in_stock TINYINT(1) NOT NULL, /* boolean */
  sku VARCHAR(255) NOT NULL UNIQUE
);
```

Relationships/Associations

- Imagine your application has users and users have profiles
- You store the user profile information in a separate table to reduce the number of columns in your users table [1]
- You need a way to associate a profile record with a user record
- A foreign key is used to create this relationship

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

---

[1] Imagine the profiles table has several other columns such as bio, location, etc.

Relationships/Associations

```
+-------------+    +---------------------------+
| users       |    | profiles                  |
+-------------+    +---------------------------+
| id | name   |    | id | user_id | photo      |
+-------------+    +---------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg  |
|  8 | Ned    |    |  2 | 9       | ned.png    |
|  9 | Sally  |    |  3 | 8       | s1.png     |
+-------------+    +---------------------------+
```

```
SELECT * FROM profiles WHERE profiles.user_id = 7;
```

```
+---------------------------+
| id | user_id | photo      |
+---------------------------+
|  1 | 7       | sally.jpg  |
+---------------------------+
```

Foreign Keys

You've seen that foreign keys can be used to associate data from one table to data from another table. Such as a specific profile record to a specific user record.

Foreign keys are more than just a column in one table referencing a row/record in another table.

Foreign keys enforce **constraints** and provide data integrity.

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

Foreign Keys

The foreign key constraint on the `user_id` column guarantees that no profile record can be inserted with a user_id value that does not exist.
That would cause the database to contain invalid data.

- We can't have a profile tied to a non existing user
- This is referred to as a referential constraint
- Referential constraints keep the database consistent

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

Foreign Keys

Example constraint violation

This is what happens when you try to insert a profile record with a non existing user ID.

```
Cannot add or update a child row:
a foreign key constraint fails
(`nycda`.`profiles`, CONSTRAINT `profiles_ibfk_1` FOREIGN KEY (`user_id`) REFERENCES `users` (`id`))
```

Foreign Keys

- Referential constraints are even more important when it comes to deleting records
- If I delete Sally (user ID 7), is Sally's profile record still relevant?
  The answer is probably no. Foreign key constraints can be configured to "cascade delete".
- You can have the profile record automatically deleted when the related user record is deleted
- The profile record is not used by any other table. It has a direct 1-to-1 relationship with a user.
  (there is no point in keeping it)

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

Database Relationships: 1-to-1

- In a one-to-one relationship, a record in one table is related to only one record in another table
- Examples:
  One `users` record has one `profiles` record
  One `customers` record has one `customer_details` record
  One `citizens` record has one `social_security_numbers` record

Database Relationships: 1-to-1

Implementing a 1-1 relationship

1. Add a column to one of the tables being linked together that **references** the **PRIMARY ID** of the other table
2. Add a **FOREIGN KEY CONSTRAINT** to the table with the column referencing the **PRIMARY ID** of the other table

In this case the `profiles` table will have a `user_id` column that references the `id` column on the `users` table.

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

Database Relationships: 1-to-1

A true 1-to-1 relationship means that every user can have no more than one profile!

```
+--------------+    +--------------------------+
| users        |    | profiles                 |
+--------------+    +--------------------------+
| id | name    |    | id | user_id | photo     |
+--------------+    +--------------------------+
|  7 | Sally   |    |  1 | 7       | sally.jpg |
|  8 | Ned     |    |  2 | 9       | ned.png   |
|  9 | Sally   |    |  3 | 8       | s1.png    |
+--------------+    +--------------------------+
```

Quiz

What constraint would we use to ensure that the profiles table doesn't have more than one row/record with the same user_id value?

```
+--------------+    +--------------------------+
| users        |    | profiles                 |
+--------------+    +--------------------------+
| id | name    |    | id | user_id | photo     |
+--------------+    +--------------------------+
|  7 | Sally   |    |  1 | 7       | sally.jpg |
|  8 | Ned     |    |  2 | 9       | ned.png   |
|  9 | Sally   |    |  3 | 8       | s1.png    |
+--------------+    +--------------------------+
```

Answer

A UNIQUE constraint

No 1-to-1 relationship is truly 1-to-1 without a UNIQUE constraint on the foreign key column.

```
CREATE TABLE users (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE profiles (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id INT(11) UNIQUE, /* References users.id UNIQUE */
    photo VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE /* FOREIGN KEY CONSTRAINT */
);
```

```
+-------------+    +--------------------------+
| users       |    | profiles                 |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | photo     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | sally.jpg |
|  8 | Ned    |    |  2 | 9       | ned.png   |
|  9 | Sally  |    |  3 | 8       | s1.png    |
+-------------+    +--------------------------+
```

Database Relationships: 1-to-many

- In a one-to-many relationship, a record in one table is related to many records in another table

- Examples:
  One **users** record has many **blog_posts**
  One **blog_posts** record has many **comments**
  One **users** record has many **tweets**

Database Relationships: 1-to-many

- Users can have many blog posts
- Ned has two blog posts
- Sally has two blog posts

```
+-------------+    +--------------------------+
| users       |    | blog_posts               |
+-------------+    +--------------------------+
| id | name   |    | id | user_id | title     |
+-------------+    +--------------------------+
|  7 | Sally  |    |  1 | 7       | foo       |
|  8 | Ned    |    |  2 | 7       | hello     |
|  9 | Bob    |    |  3 | 8       | bar       |
+-------------+    |  4 | 8       | baz       |
                   +--------------------------+
```

Database Relationships: 1-to-many

1-to-many is implemented the same way as a 1-to-1 but without the UNIQUE constraint on the FOREIGN KEY column.

1. Add a column to one of the tables being linked together that **references** the **PRIMARY ID** of the other table
2. Add a **FOREIGN KEY CONSTRAINT** to the table with the column referencing the **PRIMARY ID** of the other table

Database Relationships: 1-to-many

1. Add a column to one of the tables being linked together that **references** the **PRIMARY ID** of the other table
2. Add a **FOREIGN KEY CONSTRAINT** to the table with the column referencing the **PRIMARY ID** of the other table

```
CREATE TABLE users (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE blog_posts (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    user_id INT(11) NOT NULL, /* References users.id NOT UNIQUE */
    title VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE /* FOREIGN KEY CONSTRAINT */
);
```

```
+-------------+    +-------------------------+
| users       |    | blog_posts              |
+-------------+    +-------------------------+
| id | name   |    | id | user_id | title    |
+-------------+    +-------------------------+
|  7 | Sally  |    |  1 | 7       | foo      |
|  8 | Ned    |    |  2 | 7       | hello    |
|  9 | Bob    |    |  3 | 8       | bar      |
+-------------+    |  4 | 8       | baz      |
                   +-------------------------+
```

Quiz

What is the difference between a 1-to-1 and 1-to-many relationship when it comes to CONSTRAINTS?

Answer

What is the difference between a 1-to-1 and 1-to-many relationship when it comes to CONSTRAINTS?

1-to-1 has a UNIQUE constraint on the column referencing the other table, 1-to-many does not

Structurally, there is no other difference.

Database Relationships: many-to-many

- In a many-to-many relationship, a record in one table is related to many records in another table and vice versa.
- Examples:
  A student can take multiple classes
  A class can have multiple students

  Orders can have many Products
  Products can belong to many Orders

Database Relationships: many-to-many

Implementing a many-to-many relationship involves a join table. Also known as a pivot table.

The pivot table will associate records together from two separate tables.

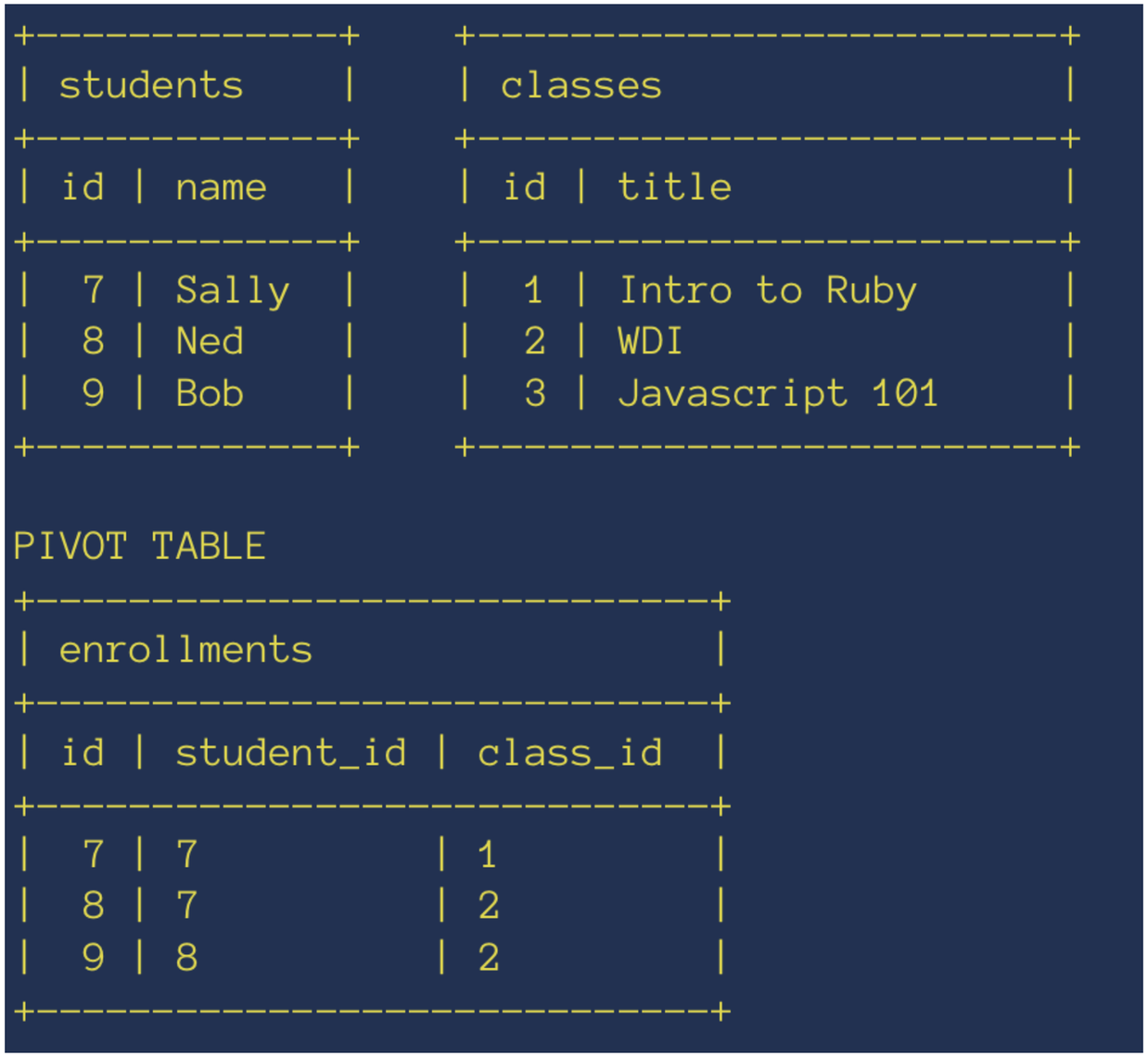
```
+-------------+    +-------------------------+
| students    |    | classes                 |
+-------------+    +-------------------------+
| id | name   |    | id | title              |
+-------------+    +-------------------------+
|  7 | Sally  |    |  1 | Intro to Ruby      |
|  8 | Ned    |    |  2 | WDI                |
|  9 | Bob    |    |  3 | Javascript 101     |
+-------------+    +-------------------------+

PIVOT TABLE
+----------------------------+
| enrollments                |
+----------------------------+
| id | student_id | class_id |
+----------------------------+
|  7 | 7          | 1        |
|  8 | 7          | 2        |
|  9 | 8          | 2        |
+----------------------------+
```

Database Relationships: many-to-many

Implementing a many-to-many relationship

1. Create a pivot table
2. Add a column for each associated table that will reference the PRIMARY KEY of the associated table
3. Add a FOREIGN KEY constraint for each column referencing a PRIMARY KEY of another table

```
CREATE TABLE enrollments (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  student_id INT(11) NOT NULL,
  class_id INT(11) NOT NULL,
  FOREIGN KEY (student_id) REFERENCES students(id) ON DELETE CASCADE,
  FOREIGN KEY (class_id) REFERENCES classes(id) ON DELETE CASCADE
);
```



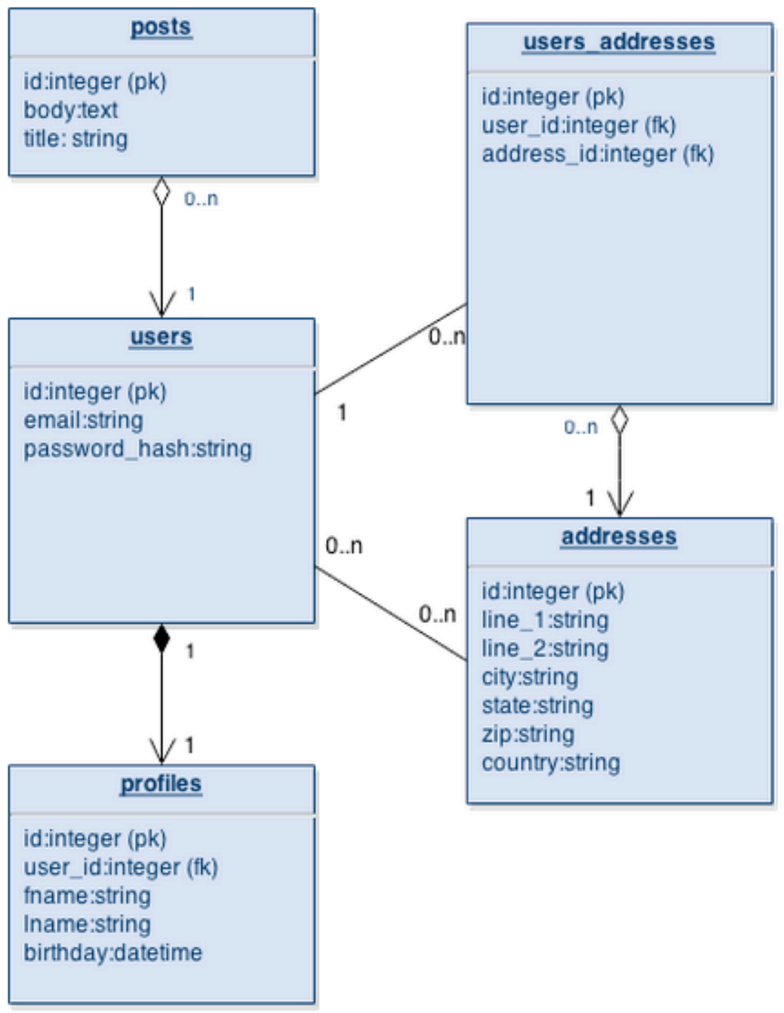
A full database diagram

- A "full" db diagram has all of your tables with their columns and data types
- The relationships between tables are illustrated with arrows

1 ---> 0..n *= one to many*

1 ---> 1 *= one to one*

0..n ---> 0..n *= many to many*



Exercise

- Let's build a database diagram for an existing major website together!

Exercise

- Develop a database diagram for one of your favorite websites.
- Start with the users table and go from there!
- Be sure to indicate the relationships between different tables in the database
- You could use www.draw.io or a pen and paper

A final word/summary

A **database** is composed of **tables**. Each **table** has **columns** and can be **related** to other tables in the database. Each column has a **data type**. Each table also has **rows** of data which correspond to the **column names** and data types described in the **schema**.

And they all lived happily ever after!

Resources

TeamTreeHouse

Database Foundations - Introduction to Data, Databases and SQL