Database Applications

Why learn SQL?

ActiveRecord and ORMs are great for making basic database queries but sometimes, depending on the app you're working on, you need to get "closer to the metal" to speed up queries

In most consumer web apps, you can get about 95% of things done with ActiveRecord, but learning a bit of SQL is great for the other 5% where the query is exotic or speed really counts.

Before we start: Installing PostgreSQL

To install PostgreSQL, try using Postgres.app:
postgresapp.com

Or follow this guide to do it from the command line with Homebrew, a package manager for Mac:
Homebrew

Windows? Use this:
Windows users

Linux: Linux users

Server vs client

The *client* sends SQL to the *server*, which runs those commands against the specified database and sends back data

The commands the server receives from the client could do many things:

- create a new database
- modify the database schema (add tables, remove tables, etc)
- create, read, update, or destroy records in a database table

Working with the psql server

If you've never booted up the psql server, you'll need to run the initdb command and specify a data directory for postgresql:

```
$ initdb /usr/local/pgsql/data/
```

The data directory is where your database, schema, and data all actually "live"!

Working with the psql server

The data directory can be in an arbitrary location but it is best to put this data directory inside the folder that postgresql resides in, in most cases the `/usr/local/pgsql` folder

If you installed psql using Homebrew, you may want to create a symbolic link between where postgresql is actually installed and the /usr/local/pgsql directory:

```
$ ln -s /usr/local/Cellar/postgresql/9.3.1/usr/local/pgsql
```

A symbolic what?

A symbolic link allows you to access a directory from elsewhere in the file system without having two different copies of it.

In Unix-based systems (Mac/Linux):

```
ln -s /path/to/existing/folder /path/to/linked/location
```

On a PC, you'll need to install a special extension:

howtogeek.com

Working with the psql server

Once the data directory issues are settled, you should be able to boot up the server:

```
$ postgres -D /usr/local/pgsql/data
>LOG:  database system was shut down at 2014-05-07 14:50:29 EDT
>LOG:  database system is ready to accept connections
>LOG:  autovacuum launcher started
```

Now you're a client!

- Once the server is booted, you'll want to start executing commands as the client to that server
- In a new terminal window, start by creating a db with the same name as the user you're currently logged in as:

```
$ createdb zachfeldman
```

- The createdb command is actually just a wrapper for the SQL code that creates a database (see the psql docs for more)

Running SQL

- Once you've created your database, you can run SQL commands against it to perform various operations (create schema, drop tables, add data etc.)
- Like many other languages you've learned, this can be done at a "prompt" line by line. To open the prompt:

```
$ psql -d postgres
```

(-d specifies your database name, postgres is the default DB)

- You can also create a script to run against the database (see next slide)

Working with psql: running a script

We can run a script directly in psql against a specific database by writing a file ending in .sql and running it using:

```
psql -d <db name> -f <filename>
```

for instance:

```
psql -d testdb -f test.sql
```

Defining a schema

- Before we can insert any data into a database, our database needs a **schema**
- The schema is a definition of all of the tables in our database
- Remember our example:
  - each table is a sheet in an spreadsheet
  - the schema is the top row of that spreadsheet
  - this top row explains the data each column will contain and its data type

Defining a schema - syntax

```
CREATE TABLE users (
   id serial PRIMARY KEY,
   fname varchar(50),
   dateCreated timestamp DEFAULT current_timestamp
);
```

If we put this into a file called test.sql and run it against a test database, it should create a table called users with an id, fname, and dateCreated column.

Syntax Breakdown:

```
CREATE TABLE[1] users[2] (
 id[3] serial[4,5] PRIMARY KEY[6],
 fname[3] varchar[5] (50)[7],
 dateCreated[3] timestamp[5] DEFAULT current_timestamp[8]
);
```

1. The create table SQL command
2. The name of the table to be created
3. Column names of the table
4. The serial data type, which will auto-increment a number for each new record
5. Column data types (`serial, varchar, timestamp`)
6. Indicates this column is the primary key, or unique identifier for records
7. The maximum length of this varchar, 50 characters
8. The default value for this column is the current time

Exercise

- Create a new database from the command line
- Define a simple schema for a users table on this database and run it against the database
- When you're done, try making a new database and a new schema with different data types - see the psql documentation for what types are available!

CRUD

Create, Read, Update, and Destroy in SQL

Interactivity!

Try running these commands as we go.

Remember that to open a console directly into your database, you can run:

```
$ psql -d <your database name>
```

from bash (the Terminal)

Creating a record

To create a new record in the users table:

```
INSERT INTO users (fname) VALUES (‘brian’);

INSERT INTO <tablename> (<column names>) VALUES (<corresponding values for this record>);
```

Creating a record

To create a new record in the users table with more than one column:

```
INSERT INTO users (fname, lname) VALUES (‘brian’, ‘fountain’);
```

Get records (read)

To get all of the records currently in the database, use SELECT *

```
SELECT * FROM users;
```

To be more, well, selective, use WHERE

```
SELECT * FROM users WHERE fname = 'brian';
```

Get records (read)

To only retrieve specific columns:

```
SELECT (fname) FROM users;
```

You can also use WHERE with this style of query:

```
SELECT (fname) FROM users WHERE fname = 'brian';
```

Update records

To update every single record:

```
UPDATE users SET fname = 'john';
```

To update a specific record:

```
UPDATE users SET fname = 'john' WHERE fname = 'brian';
```

Destroy records

BE CAREFUL. There is no UNDO in database-land!

```
DELETE FROM users WHERE id=1;
```

To delete all records from a table:

```
DELETE FROM users;
```

Exercise

Take one of the database diagrams we created in class today and attempt to implement it using CREATE TABLE to create all of the necessary tables: in a single file.

Run that file against a test database and perform at least 20 test queries to make sure that all of your tables were set up correctly.

Advanced Exercise

Backticks are used to run command line commands in a Ruby script.

```
irb(main):002:0> cmd = `pwd`
=> "/Users/fountain\n"
```

Create a Ruby script that uses backticks to run queries against your database. An example line of code that should be helpful:

```
cmd =`psql -d testtwo -c "SELECT * FROM USERS"`
```