SQL

Overview

The primary focus of this class should be the syntax of SQL and how to interact (using CRUD operations) with a database persistence layer in the browser. While we will be using the html5sql.js library to interact with our database, we also have included an abstraction layer, webDB, to make it easier for students to perform CRUD operations and focus on structuring meaningful queries. Students will need a primer on how this abstraction layer works, but should be more focused on SQL itself - be sure to discuss this prior to lab so that students know that they are welcome to ask questions about the wrapper library.

Learning Objectives

Understand the basic concepts of a database Effectively use basic SQL commands to create, read, update, and delete rows from a table

- Topic 1 - CRUD
    - CRUD - Databases as a resource
        - C - CREATE
        - R - READ
        - U - UPDATE
        - D - DESTROY (or DELETE)
    - Relational Databases
        - Discuss differences between document based storage
        - Discuss common DBMS - MySQL, Postgres, SQLite, etc
        - Discuss concepts of relational data
    - SQL
        - Syntax
        - Statements
        - Clauses
        - Constraints
        - Expressions
        - Predicates
        - SQL cheat sheet
    - Data types
        - INTEGER
        - NOT NULL
        - PRIMARY KEY
        - CHAR
        - VARCHAR
    - WebSQL
        - Discuss importance of using this for exposure and teaching concepts, not for large scale application development
        - Discuss the webDB abstraction layer
            - webDB.execute method
                - Discuss different options
                - Discuss dynamic value based data injection to avoid SQL injection
            - webDB.init method

            - Discuss how it connects to the database
        - Discuss
- Demo - How
    - Demo SQL interaction in Chrome Dev Tools
        - Initialize a connection to the database
        - Create a test table
        - Use SQL to interact with the database in the Chrome database console
        - Insert records from sample database
        - Delete records from sample database
        - Update records from sample database
        - Demonstrate how to drop a table
        - Use different clauses to demonstrate SQL syntax
        - Discuss importance of using a library or abstraction layer to communicate with the database through the use of JavaScript
        - Exposure to formation of more advanced SQL queries.
        - Knowledge of how to interact with WebSQL in the browser.
        - Ability to interact with Web SQL using JavaScript

DEMO - Intro to CRUD

Adding Libraries

```
<!-- DONE: Include the script files we need, that give us an API to the WebSQL feature of
Chrome: -->
<script src="vendor/scripts/html5sql.js"></script>
<script src="scripts/webdb.js"></script>
<script src="scripts/person.js"></script>
```

`html5sql.js` is a light JavaScript module that makes working with the HTML5 Web Database a whole lot easier. Its primary function is to provides a structure for the SEQUENTIAL processing of SQL statements within a single transaction. This alone greatly simplifies the interaction with the database however it doesn't stop there. Many other smaller features have been included to make things easier, more natural and more convenient for the programmer.

`webdb.js` is a javascript and jQuery plugin which let's you play with WebSQL feature of HTML5 easily. You can create Web Databases and then do anything with them.

CREATE

DATA

```
[
  {
    "id" : 1001,
    "first" : "Edgar",
    "middle" : "Allan",
    "last" : "Poe",
    "dob" : -5079081600000,
    "bio" : "Born January 19, 1809, Boston, Massachusetts"
  },
  {
    "id" : 1002,
    "first" : "Roger",
    "middle" : "Allan",
    "last" : "Poe",
    "dob" : -5079086500000,
    "bio" : "Born January 19, 1819, Boston, Massachusetts"
  }
]
```

CODE

```
  // DONE: Set up a DB table for person.
  Person.createTable = function(callback) {
    webDB.execute(
      // what SQL command do we run here inside these quotes?
      'CREATE TABLE IF NOT EXISTS people (' +
          'id INTEGER PRIMARY KEY, ' +
          'first VARCHAR(255) NOT NULL, ' +
          'middle VARCHAR(255), ' +
          'last VARCHAR(255) NOT NULL, ' +
          'dob DATETIME, ' +
          'bio TEXT NOT NULL);',
      function(result) {
        console.log('Successfully set up the person table.', result);
        if (callback) {
          callback()
        }
      }
    );
  };
```

INSERT

```
  // DONE: Insert an article instance into the database:
  Person.prototype.insertRecord = function(callback) {
    console.log("******** insert Record *******");
    webDB.execute (
        [
            {
              sql: 'INSERT INTO people (id, first, middle, last, dob, bio) VALUES (?, ?, ?, ?,
?, ?);',
              data: [this.id, this.first, this.middle, this.last, this.dob, this.bio]
            }
        ],
    callback);
  };
```

READ

```
 Person.fetchAll = function(callback) {
  webDB.execute('SELECT * FROM people', function(rows){
    if (rows.length !== 0){
        console.log(rows);
        Person.loadAll(rows);
        callback();
      } else {
        $.getJSON('data/people.json', function(rawData){
          console.log('why it');
          var people = rawData.map(function(item){
            var person = new Person(item);
            person.insertRecord();
            return person;
          });
          Person.loadAll(people);
          callback();
        });
      }
    });
  };
```

Small refactor of function constructor

```
  // DONE: Refactor to expect the raw data from the database, rather than localStorage.
  Person.loadAll = function(rows) {
    Person.all = rows.map(function(ele) {
      return new Person(ele);
    });
  };
```

AJAX + CRUD

```
Person.fetchAll = function(callback) {
  webDB.execute('SELECT * FROM people', function(rows){
    if (rows.length !== 0){
      console.log(rows);
      Person.loadAll(rows);
      callback();
    } else {
      $.getJSON('data/people.json', function(rawData){
        console.log('why it');
        var people = rawData.map(function(item){
          var person = new Person(item);
          person.insertRecord();
          return person;
        });
        Person.loadAll(people);
        callback();
      });
    }
  });
};
```

DEMO SQL COMMANDS

READ `SELECT * FROM people`

INSERT `INSERT INTO people (id, first, middle, last, dob, bio) VALUES (1003, "Cesar", "Rafael", "Jimenez", -5079081600000, "He is cool");`

DELETE `DELETE FROM people WHERE id=1` UPDATE `UPDATE people SET first="Cesar" WHERE id=1001;` DROP `DROP TABLE books`

DEMO - Pair Programming

Example #1

```
<!--STEP #1-->
<!-- J-D: Include the script files we need, that give us an API to the WebSQL feature of
Chrome: -->
<script src="vendor/scripts/html5sql.js"></script>
<script src="scripts/webdb.js"></script>
<script src="scripts/article.js"></script>
<script src="scripts/articleView.js"></script>
<script>
```

Example #2 - CREATE

```
// J+D: Set up a DB table for articles.
Article.createTable = function(callback) {
  webDB.execute(
    'CREATE TABLE IF NOT EXISTS articles (' +
    'id INTEGER PRIMARY KEY, ' +
    'title VARCHAR(100) NOT NULL,' +
    'category VARCHAR(100),' +
    'author VARCHAR(100),' +
    'authorUrl VARCHAR(2083),' +
    'publishedOn VARCHAR(10) NOT NULL,' +
    'body TEXT NOT NULL);',
    function(result) {
      console.log('Successfully set up the articles table.', result);
      if (callback) {
          callback();
      }
    }
  );
};
```

Example #3 - TRUNCATE

```
// J+D: Use correct SQL syntax to delete all records from the articles table.
Article.truncateTable = function(callback) {
  webDB.execute(
    'DELETE FROM articles;', // <----finish the command here, inside the quotes.
    callback
  );
};
```

Example #4 - DELETE

```
// J+D: Delete an article instance from the database:
Article.prototype.deleteRecord = function(callback) {
  webDB.execute(
    [
      {
        'sql': 'DELETE FROM articles WHERE id=?;',
        'data': [this.id]
      }
    ],
    callback
  );
};
```

Example #5 - UPDATE

```
  // J+D: Update an article instance, overwriting it's properties into the corresponding record
in the database:
  Article.prototype.updateRecord = function(callback) {
    webDB.execute(
      [
        {
          'sql': 'UPDATE articles SET title=?, category=?, author=?, authorUrl=?, publishedOn=?
body=? WHERE id=?;',
          'data': [this.title, this.category, this.author, this.authorUrl, this.publishedOn,
this.body, this.id]
        }
      ],
      callback
    );
  };
```

Example #6 - CRUD + AJAX

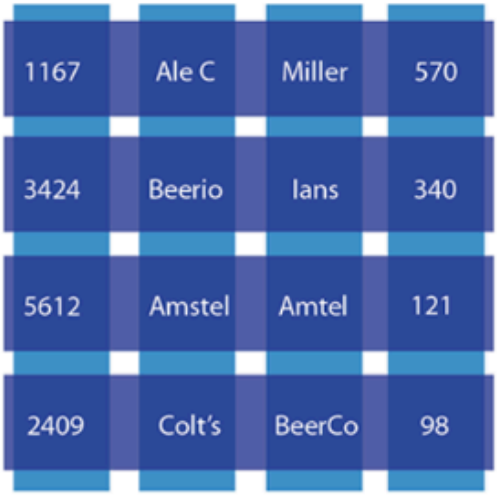
```
  // J+D: Refactor this to check if the database holds any records or not. If the DB is empty,
  // we need to retrieve the JSON and process it.
  // If the DB has data already, we'll load up the data (sorted!), and then hand off control to
the View.
  Article.fetchAll = function(next) {
    webDB.execute('SELECT * FROM articles', function(rows) { // J+D: fill these quotes to
'select' our table.
      if (rows.length) {
        Article.loadAll(rows);
        next();
        // J+D: Now, 1st - instanitate those rows with the .loadAll function,
        // and 2nd - pass control to the view by calling whichever function argument was passed
in to fetchAll.

      } else {
        $.getJSON('data/hackerIpsum.json', function(rawData) {
          // Cache the json, so we don't need to request it next time:
          rawData.forEach(function(item) {
            var article = new Article(item); // Instantiate an article based on item from JSON
            // J+D: Cache the newly-instantiated article in the DB: (what can we call on each
'article'?)
            article.insertRecord();

          });
          // Now get ALL the records out the DB, with their database IDs:
          webDB.execute('SELECT * FROM articles', function(rows) { // J+D: select our now full
table
            // J+D: Now, 1st - instanitate those rows with the .loadAll function,
            // and 2nd - pass control to the view by calling whichever function argument was
passed in to fetchAll.
            Article.loadAll(rows);
            next();
          });
        });
      }
    });
  };
```

Difference between Document Storage and Relational Databases

In a relational database system you must define a schema before adding records to a database. The schema is the structure described in a formal language supported by the database and provides a blueprint for the tables in a database and the relationships between tables of data. Within a table, you need to define constraints in terms of rows and named columns as well as the type of data that can be stored in each column.

In contrast, a document-oriented database contains documents, which are records that describe the data in the document, as well as the actual data. Documents can be as complex as you choose; you can use nested data to provide additional sub-categories of information about your object. You can also use one or more document to represent a real-world object. The following compares a conventional table with document-based objects:

Beers Table



Beer Documents