Databases

Bootcamp #8 - Day One

Topics Covered

- Learn how to connect to a database
- Learn how to perform DML and DDL operations on database
- Learn how to apply various join types on database tables.
- Learn how to write/execute Stored Procedures
- Learn about Transact-Sql with examples
- Learn how to use ORM’s for data mapping
- Discuss about MySql vs NoSql databases
- Demonstrate how to transform data returned from database for use in an application in your preferred programming language

What are databases?

- The long term memory of a system
- There’s lots of different implementations, but two main types
  - Relational, e.g. mysql, mssql
  - Non-Relational, e.g. couchbase, mongodb
- The many choices reflect the consistently high importance of databases in the wide variety of projects that involve technology.

Common Terms

Table: A collection of records. E.g. Customers.

Columns (or fields): Each table has its own columns. A column is a defined property that a record can have. Columns have types (int, datetime, etc.) and constraints (unique, auto increment, etc.)

Rows (or records): A row represents an individual item of data with it’s own values for each of the columns in a table. E.g. Customer #137 has 4 friends and signed up on 2014-12-25.

Identity (id): A row usually has a unique identity column that can be used to refer to it in queries or code.

Object Modelling

- Useful for planning what data will look like in your database

Object Name
Field1 : type Field2 : type

Cheese
Name : string Tasty : bit Type : string

Exercise #1: Connecting

You’ll need a database driver - a piece of software that your application will use to talk to the database. Java typically uses JDBC drivers. For dotnet there will be plenty of nuget packages for each database type.

To complete this exercise you just need to be able to show that you’ve connected using a visual designer tool **MySql Workbench** to the following database:

Host: db-workshop.public-dev.zuto.cloud
Username: db-bootcamp-ro
Password: ElephantArchaeologist
Default Schema: readonly

(Individual Credentials will be provided)Work in pairs!

Structured **Q**uery **L**anguage

SQL has two main subsets of commands:

1. Data Definition Language - This is used to define how the database is structured; creating and altering tables, etc.
2. Data Manipulation Language - This is used to work with the actual data stored inside the database; creating, retrieving, updating and deleting content.

For some reason the syntax IS VERY SHOUTY. It doesn't have to be, it works just the same in lowercase, but the majority of SQL you'll see will be WRITTEN LIKE THIS.

Retrieving Data from a table

SELECT *field1, field2*
FROM *table*

You can also use * to retrieve all fields instead of listing them individually

You can filter data using a WHERE clause

SELECT *field1, field2*
FROM *table*
WHERE *id_field* = 321

And you can set the order that results come back in

SELECT *field1, field2*
FROM *table*
ORDER BY *field3*

Exercise #2: Getting data out

In the database you've just connected to, there's a table named 'Cheeses'

For to complete this exercise you just need to retrieve all the cheeses in the table and print out their name and type.

Updating Data

UPDATE *table*

SET *field1 = value, field2 = value*

Again we can use a WHERE clause to limit the scope of what we’re doing.

UPDATE *table*

SET *field1 = value, field2 = value*

WHERE *id_field = 321*

Exercise #3: Changing data

Now we're going to alter the data in our database table. To prevent confusion every pair will get their own user credentials that only has access to its own schema.

Once you've connected to the database with your new credentials, you'll have your own copy of the Cheeses table that you can alter.

Try changing the 'Tasty' column from true to false for the Stilton record.

Adding new data

```
INSERT INTO table(string_field, number_field)
VALUES ('value1', 5)
```

If some of the fields on a table are optional then you can skip them in the list and not provide a value.

You can add multiple records at a time:

```
INSERT INTO table(string_field, number_field)
VALUES ('value1', 5), ('value2',42), ('other',0)
```

Exercise #4: Adding our own records

As our *fromagerie* grows in popularity we'll need more and more product types to keep our customers happy.

To complete this exercise add a new cheese to your Cheeses table.

N.B. You don't need to provide an id for a new cheese - the table knows to generate the id itself (this is a setting on the table).

Removing data

DELETE FROM *table*

This removes all data in the table! Combine with a WHERE clause to surgically remove records:

DELETE FROM *table*
WHERE *field* = *value*

Exercise #5: Deleting records

Apparently too much choice can be a bad thing.

To complete this exercise, remove some of the cheeses in the table until there's only 4 left.

Normalisation

- Our data is ‘flat’. Everything is stored in one table
- We’ve got repetition about sellers of Cheeses
- It’s not so bad when there’s only a few cheeses, but imagine having to update every record in a large table because the sellers phone number changed!
- Normalisation is the process of transforming a database into ‘normal’ form

Example Normalised Data

Id	Name	Employer
1	Mim	1
2	Rob	1
3	Sue	2

Id	Company	Building
1	BBC	A
2	ITV	B
3	Sky	B

Id	Building	Postcode
A	Media City	M50 2BH
B	TV Town	AB12BC

Creating Tables

CREATE TABLE *table_name* (

field1 data_type,

field2 data_type

)

If you want to have an auto-incrementing identity column:

CREATE TABLE *table_name (*

id_name int NOT NULL AUTO_INCREMENT

...*other fields*...

PRIMARY KEY (*id_name*)

)

Exercise #6: Normalising Cheeses

Creating Tables

Extract information about the seller of cheese into a separate database.

You'll need to create a new table to store the extracted information as well as adding a new column to the cheeses table to reference the new sellers table.

Joining related tables

SELECT *table1.field1, table2.field2*

FROM *table1*

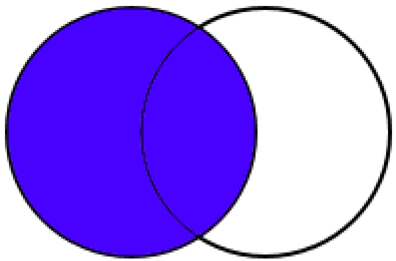
INNER JOIN *table2* ON *table1.field2 = table2.id_field*

Again, you can use * to get everything from both tables, or you can do *table1.** to get everything from a particular table.

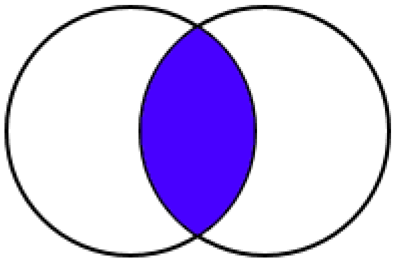
There are different types of joins. Depending on which one you use the query returns different data when there's no matching records

Types of Join

LEFT JOIN



INNER JOIN



Plus a few others

Exercise #7: Joins

Write a select query that lists all the cheeses where the seller is located in ‘Manchester’*

*If you’ve ended up with no sellers in Manchester then use a value that you know will have results

Stored Procedures

- Stored procedures are queries stored in the database
- They can be used to run CRUD operations as well as more complicated queries.
- They can combine multiple queries into one call (e.g. Create a new record then update another with a reference to the new record)
- Well named Stored Procedures can give context when investigating database performance/issues.
- They can be combined with permissions to restrict what connecting users can and cannot change in the database.
- In early databases Stored Procedures were quicker than dynamic queries constructed in code.

Example Stored Procedure

```
CREATE PROCEDURE `create_tasty_cheese` (IN cheese_name varchar(255))
BEGIN
INSERT INTO Cheeses(name, tasty) VALUES(cheese_name, 1);
SELECT SUM(tasty) FROM Cheeses;
END
```

And then:

```
CALL create_tasty_cheese(‘cheese++’)
```

Transactions

Transactions allow us to run many SQL commands together and only make changes if all of the commands are successful.

START TRANSACTION;
INSERT INTO *table1* …;
INSERT INTO *table2*…;

When writing stored procedures or multiple queries in mysql, each operation needs to end with a semicolon.

Then:

COMMIT;
Or to revert the changes:
ROLLBACK;

Exercise #8: Transactions

Using MySQL workbench explore transactions by making changes to your data, running queries and then seeing what happens when you commit or rollback the transaction.

Security

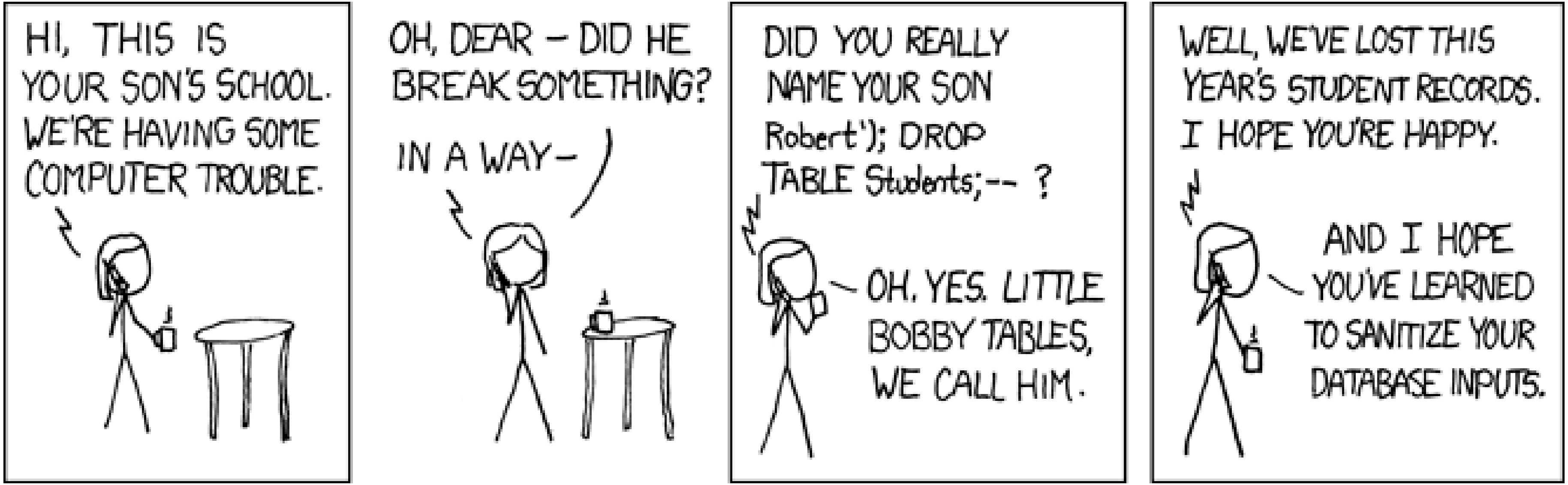
Imagine you’re writing code to log in to a website. You take their username and password and could end up with SQL like this:

SELECT id FROM users WHERE email = ‘their_email’ AND password = ‘their_password’

This is vulnerable to SQL Injection - if I input my password as ‘ OR ‘1’=’1’ # then the SQL ends up like this:

SELECT id FROM users WHERE email = ‘their_email’ AND password = ‘’ OR ‘1’=’1’ #’

("Manchester")+("Digital")>



https://xkcd.com/327/

Input Sanitisation

In order to avoid SQL injection or other unexpected behaviour, you can sanitise user inputs - either manually or using functions that come with many frameworks.

For example you could escape characters such as ‘ so that they become \’ which would stop the examples in the previous slides from working.

You can also use **prepared statements**, a feature in most database frameworks that allows you to define a query and then tell it to run with the user inputs. The framework then sanitises the input for you.

Non-Relational Databases

Can store data in a variety of ways but any relationship between the data is not stored with it. You might choose to use a different database type if you have concerns about speed or scalability.

Examples:

Key-Value databases (Redis, etc.) store data without defined structure against ids.

Document databases (Couchbase, DocumentDB, etc.) are a subclass of Key-Value databases where you can store entire blocks of JSON/XML.

Object-Relational Mapping (ORM)

ORM frameworks exist that allow us to not write the same SQL over and over again, converting objects to SQL and results back to objects. By using such a framework you can end up (after defining your object and mappings) that looks like this:

```
session.save(new Cheese(‘dairylee’,true));
```

Example frameworks: Hibernate (java), Entity (c#)

Tomorrow

Putting all of this together to build a website that can store and retrieve data