# Semi-Structured data and Advanced Data Modelling CW2

## Assumptions

*Below is a list of the assumptions we considered when designing this system.*

1. For Drivers, we assume that employment records consist of their hire data and contract type
2. Driver contract type is part of the driver entity.
3. Drivers and operators are all considered employees'. There can be at most 8 operators.
4. An employee must have at least 1 shift in the system.
5. Bookings consist of a booking ID, Date, time, start address, customer name, telephone, number of passengers.
6. Each booking can have multiple stops which will be a separate collection linking to bookings via the booking ID.
7. There are two types of cars driven by drivers, regular 5 seaters' (4 passenger max) and 7 seaters' (6 passenger max).
8. For payments, customers can pay over the phone by card or by cash during the journey.
9. Each journey can only be paid by one method (card or cash)
10. Client regular bookings can be done daily, weekly and monthly.
11. A client must have at least one regular booking in the system. (otherwise they wouldn't be registered)
12. Drivers pay the cab company weekly and therefore pay periods are on a weekly basis.
13. The pay periods follow the tax year (6$^{th}$ April). Revenue and payslips are collected/given every Friday.
14. People who live at the same address may both work for the company.
15. There can be multiple drivers on a shift at the same time.

## Collections

*Below is a description of each of the collections used explaining how they are modelled*

**drivers**

This collection stores all information about the drivers in the taxi company. It contains the bulk of the data. Each document stored contains all the information relating to a driver, which consists of their name, address, telephone, hire date, contract type, shifts, car and revenue. The collection uses embedded documents to store the address, shift, car and revenue information to allow the simulation of relations (as MongoDB is document oriented).

**operators**

Similarly, to drivers, this collection stores information about the operators in the company. An embedded document is again used to store the address of the operators. The attributes are the same apart from the contract type which operators don't have.

**clients**

This collection stores all information about the company's clients. This consists of their name, address, client type (corporate and private) and bookings. address and client bookings are both stored using embedded documents.

**customerbookings**

This collection contains all bookings made by regular customers (not clients). It stores the date, time, passengers, pickup address, stopping destinations and customer's name. embedded in the collection is also information about the payment for the booking.

**cars**

This information is embedded in driver. Each document contains information on the cars' registration number, age, last MOT date, current status and capacity.

**clientbookings**

This collection holds all the bookings and is embedded within clients, this collection contains information about the client bookings similarly to customer bookings with the exception of having a booking type attribute (daily, weekly or monthly). Due to this, each payment is nested within the embedded document in an array.

**payment**

This document holds all the information on payments for a booking, it is embedded within both client and customer bookings and consists of a price, payment type and status (whether the bill is paid or not).

**revenue**

this collection is stored as an embedded document within drivers. It contains information about a driver's earnings per pay period and their gross earning for that period. Each period is 2 weeks long.

**shifts**

This collection is stored within drivers and contains information about a driver's shifts. This is stored in an array as a driver can have multiple shifts. Each shift consists of a date, start time and end time.

**stoppingDestinations**

This collection is embedded within both bookings collections using an array. This is because each booking can have multiple stops along the way before they reach the final stop. It consists of embedded documents which store the addresses.

**Performance Monitoring Tools**

Monitoring MongoDB is useful to determine the current state of the database. Performance data can be used to identify abnormal behaviour and fix small issues before they get worse. It also allows the developer to test the efficiency of different queries the system will make regularly.

**Explain()**

This command can be used with the argument “executionStats” to monitor the number of documents searched when executing a query. When used on query we can see by the “docsExamined” stat how many documents were checked during the query.

When run on a simple query such as:

db.drivers.find({"contractType":"FF"}).explain("executionStats")

the number of documents examined is 10 as the query is relatively simple. Also, the data is not indexed to it search through all the documents till it finds what it needs. If the contract type was indexed, it wouldn’t need to search every document.

**Profile()**

This command tells us what was run, how many documents were scanned and how much data was returned. This is useful for monitoring the performance of more complex operations which explain doesn’t monitor.

When used on this query:

```
db.clients.aggregate([
        {
                $project:           {
                        bookings: {
                                $filter: {
                                        input:"$bookings",
                                        as:"a",
                                        cond: { $gt: [ "$$a.payment.price", 40 ] }
                                }
                        }
                }
        }
])
```

Running db.system.profile.find() shows us how many documents were scanned at each section of the query where it entered the arrays to search for the bookings costing over £40. This would take longer if the data set was larger as there are more steps involved and many more documents scanned. The profile command will allow us to see these differences and change the structure of the database.

# UML Diagram showing the documents in our design and how they relate to each other