

MYSENSEI

BLIV SENSEI

TILMELD

LOG IND

FIND DIN UNDERVISER I DIT NÆROMRÅDE

SÅDAN FUNGERER DET

HVAD SØGER DU?

AFSTAND KM

SØG

MYSENSEI

DATABASE DESIGN

Brian Munksgaard | Databases | December 9, 2015

Contents

Introduction.....	2
Scenarios	2
Entities.....	3
Database design.....	4
Relational Model	4
<i>Conceptual Model</i>	4
<i>Physical Model</i>	6
<i>Document Model</i>	7
Database Implementation	8
Microsoft SQL Server	8
<i>Tables and Constraints</i>	8
<i>Stored Procedures</i>	10
<i>Views</i>	13
MongoDB	16
<i>Group Hierarchy</i>	16
<i>CourseOverview</i>	16
Late changes.....	18
Missing Attributes	18
<i>Relational Database Changes</i>	18
<i>MongoDB Changes</i>	19
Final Notes	20
Choosing the Database	20

Introduction

The concept behind the free service MySensei is to make it easier for people seeking to learn a new skill meet with people that would like to educate others in that skills.

Instead of seeking, often expensive and more formalized, courses with private companies, the idea is, based on the assumption that people like to learn and to teach others their skills, to support the exchange of skills in very informal ways and at very low prices.

The MySensei service is an online web based marketplace where people can search for courses and/or set up courses.

One of the many important decisions, when establishing such a service, is to decide how to organize your data, how data is related and how data is to be stored. This document contains information about just that.

Scenarios

Below I have written three different scenarios. The scenarios are used to give an idea of the scope of the MySensei service. What type of persons will use it, how will they use it, what motivates them etc. The more scenarios the better and three may actually not be enough. Also, the scenarios helps identify entities and how are they related.

Scenario 1: A single mother wants to search for a guitar teacher for her 11 year old son. She wants the son, to have a one hour lesson once a week. The lessons are going to be at her place. The day and the time of day for the next lesson is discussed and agreed upon after the end of each lesson.

Scenario 2: As an experienced sail boat captain I want to teach others how to sail. My reason for this is that sail clubs often have rather expensive, long running sailing courses focusing on all the aspects of sailing including lots of theory. This may cause some people never to get started with sailing or to always sail their boat using their engine. My course should be about practical sailing and seamanship, what works and what does not work, not about reading books and talking about it. That way people get the fun stuff first and can attend a more formalized course later on. In the coming spring we will sail Saturdays from 10:00 to 13:00. In the coming summer we sail Saturdays from 10:00 to 13:00 and Wednesdays from 18:00 to 20:00.

Scenario 3: As an experienced guitarist I would like to earn a little money while studying. I prefer to teach students around own age (20-30) so we may have more than playing the guitar in common. I live and study in the city of Århus. I suggest that we meet twice a week, every Tuesday and Thursday at 18:00 to 20:00.

Entities

Given the web site profile image below:

The screenshot shows the MySensei website interface. At the top, there's a green header with the MySensei logo and navigation links: "BLIV SENSEI", "FIND EN SENSEI", and "MIN SIDE". The main content area is divided into two columns. The left column features a course titled "Hans køkken og madkunst" with a description, hashtags, and a "TAG KONTAKT TIL HANS J. PETERSEN" button. The right column shows a profile for Hans J. Petersen, including a placeholder for a profile picture, a bio, and a "Del med dine venner" section with a "Like" button and a count of 5. The course details on the left include a price of 25,- pr time, 1-3 participants, and a time slot of 17.30-20.00.

And the previously described scenarios and considering the MySensei service as a simple course booking system, the following entities have been identified:

- Person.
- Credentials.
- Course.
- Course Group.
- Course Lesson.
- Course Tag.
- Review.
- Location.
- City.

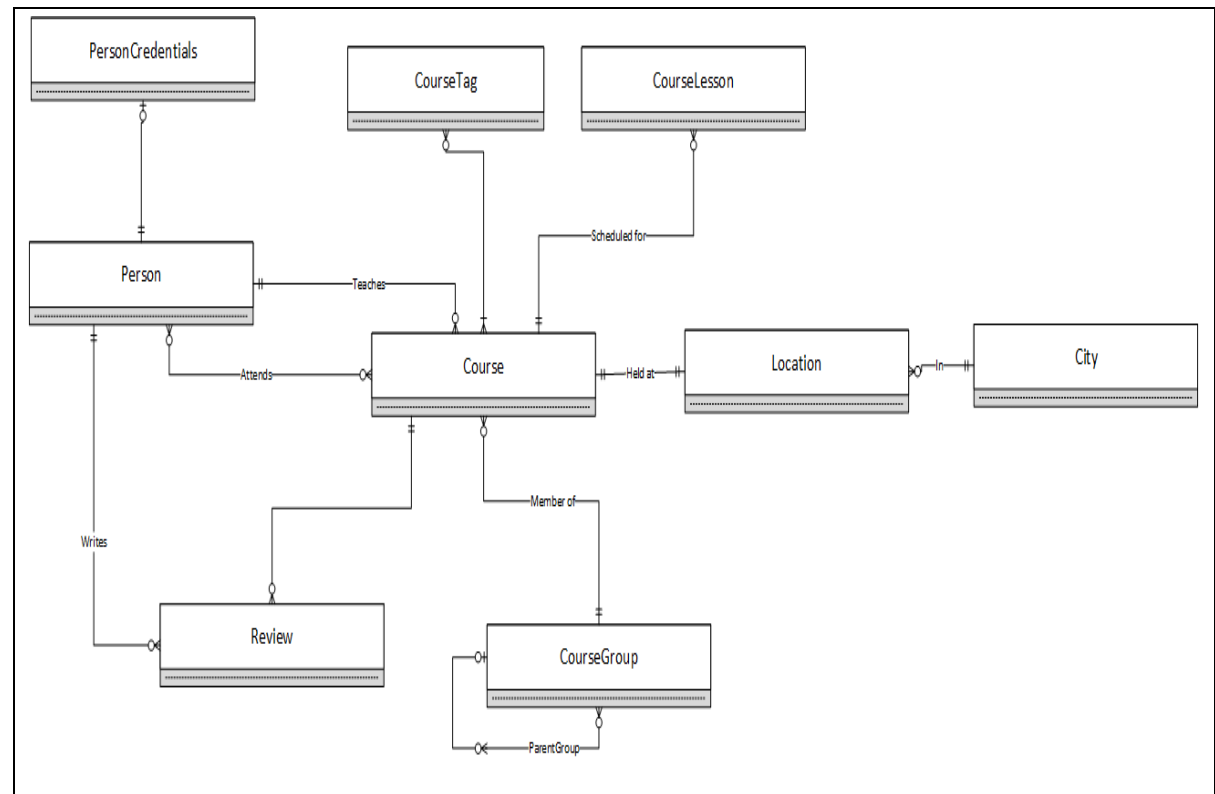
Whether or not we want to use a relational database or a document database the entities are the same. Using a relation database we will normalize the entities and relations until on the third normal form, which may lead to more entities depending on how we choose to implement relations and cardinality. Using a document database we will aggregate entities into documents of certain type and/or purpose.

Database design

RELATIONAL MODEL

Conceptual Model

Below you can see the conceptual data model with the entities that we identified previously.



From the model you can see:

- That a person may have a set of credentials and that those credentials belongs to only one user.
- That a person may attend a course and a course can have multiple attendees.
- That a person may teach at several courses but a course has only one teacher.
- That a person can be a teacher at one course but also attend another course.
- That a person may write a review and that a review is always done by one person.
- That a course may have zero or many reviews
- That a course is always a member of a course group and that a group may have a parent group. Groups should be admin defined in order to keep the structure simple and concise. If users had the options to create their own groups, I believe it would quickly become very hard keep the number of groups down and to avoid duplicate groups.
- That a course takes place at a specific location in a specific town. Location names and streets can be entered in so many different ways, that I have decided that the course “owns” the location and that a location only exists as long as the course does.
- That a course can have zero or more planned lessons.
- That a course can have multiple tags or no tags at all and that the same tag can be assigned to one or more courses. Tags are user defined, informal and more personal metadata describing a course.

The rules stated above can be implemented and enforced in a relational database system. The model is relatively small and easy to understand. If needed, we can easily extend it. Some plausible extensions are:

- A course lesson can be held at an alternative location.
- A person must also have a location.
- A person (as a teacher) can also be reviewed not just courses.
- A course request where a person can register a request for a specific course.

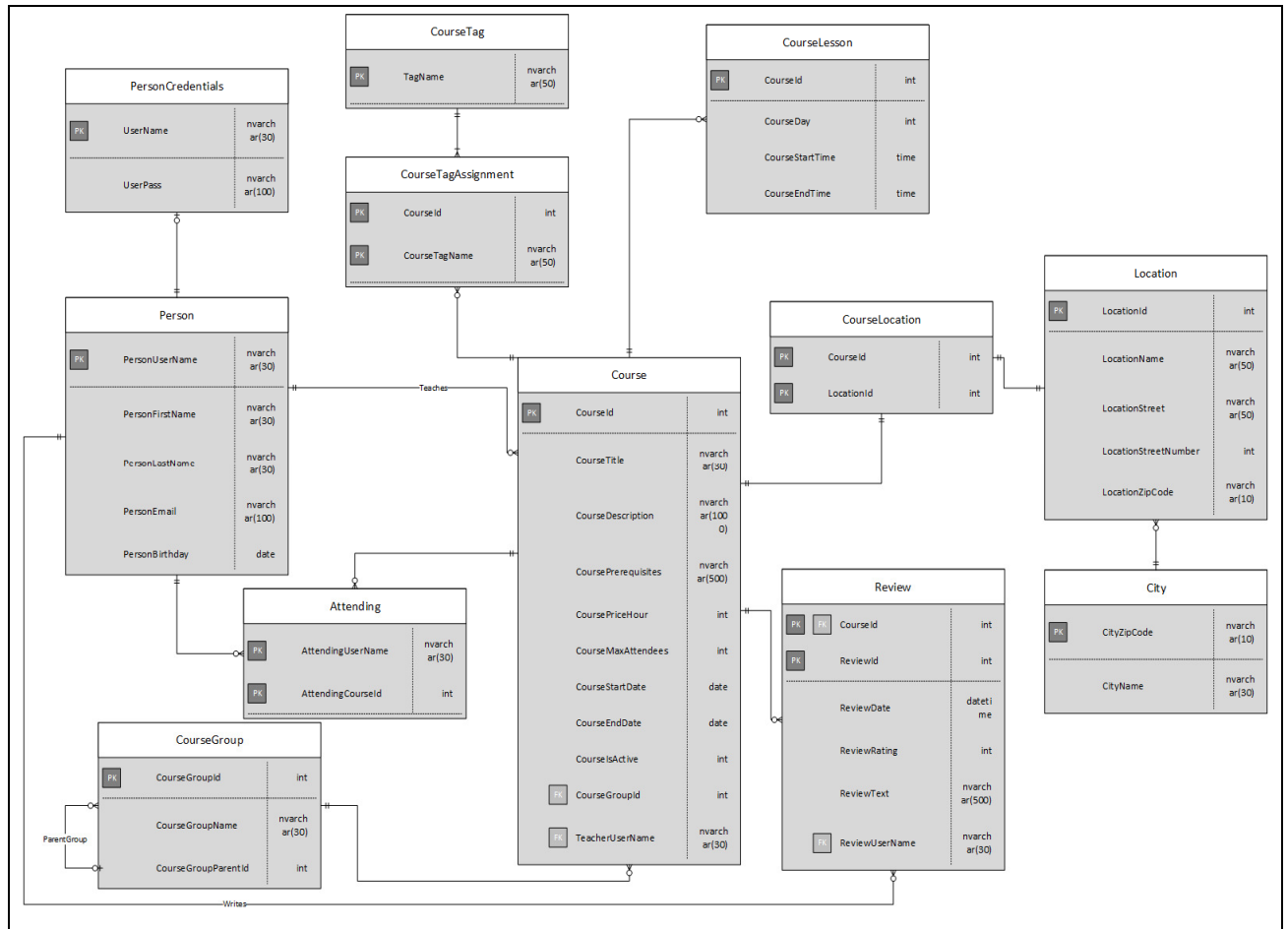
One big question is how to implement the Course and CourseLesson entities. How do we want to specify the date and time the lessons take place? I have assumed that the course entity holds information about the period (from and to date) the course is being held. The CourseLesson entity then holds information about the day(s) with each day having a start and end time. This leads to another possible extension of the model:

- A course can have exclusion dates, where lessons are not taking place.

However, I assume that teachers and students will handle cancellations manually using email or SMS. The simpler the better.

Physical Model

In the physical model attributes and their data types have been added. Since we know that the data model must be implemented in Microsoft SQL Server, the data types match those available in that RDBMS.



As you can see, the physical model has three more tables not present in the conceptual model: Attending, CourseLocation and TagAssignment. These tables are used to implement the many-to-many relations.

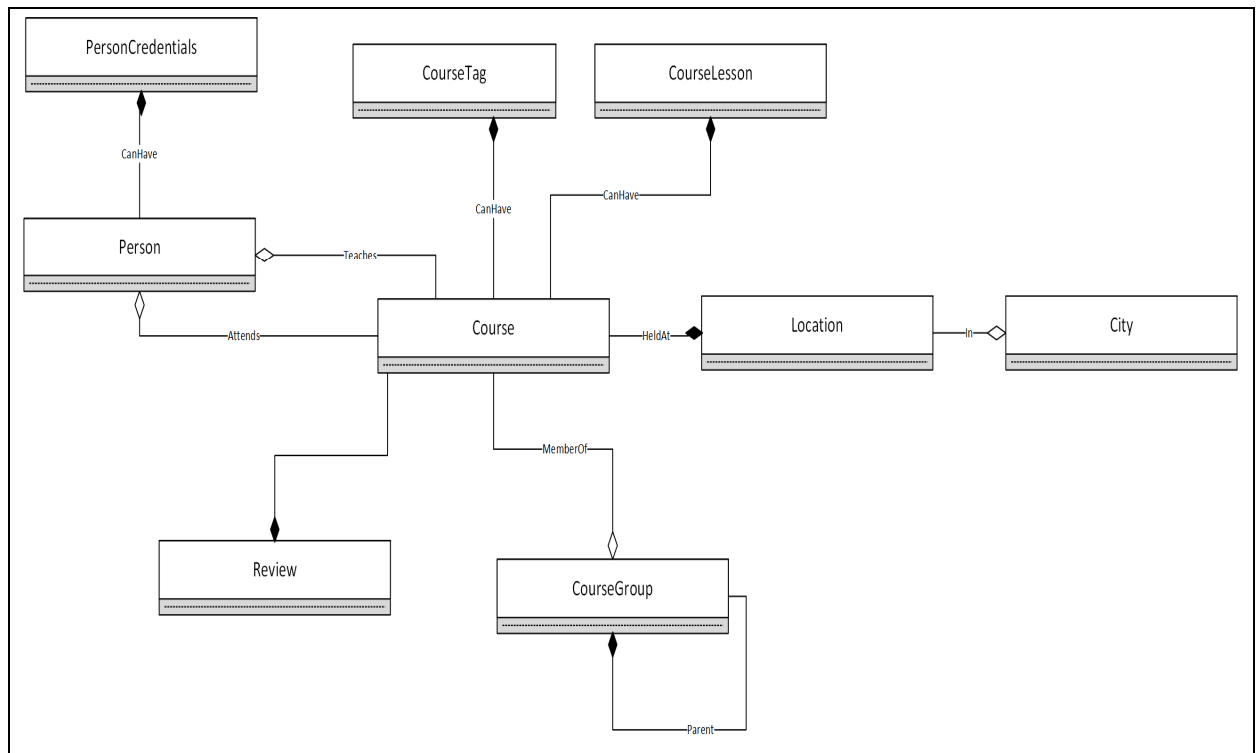
Also note the CourseIsActive attribute on the Course entity. This is a true/false field that will be implemented by using an int datatype with constraints.

Document Model

Below I have tried to create a data model that can be implemented in a document database such as MongoDB. I assume that the entities contains the same attributes as shown in the physical relational model. The purpose of the model, is to show, how the entities are aggregated.

To display entity aggregation in the document model I have used two notations from UML, the aggregation (the white diamond) and the composition (the black diamond). An aggregation is used, where the entity on the diamond end is loosely coupled and that entity can live on its own. A composition is used, where the entity on the diamond end is tightly coupled and cannot live on its own.

In the model below, for example, this means that City is not deleted when a Location is because City is loosely coupled with location. On the other hand, when a Person is deleted, PersonCredentials are also removed because those two entities are tightly coupled.



MongoDB is a so-called schema-less database. In reality, there is always some sort of schema definition. The difference is that MongoDB does not enforce schema rules as an RDMS does. The schemas you apply to the document database are merely a matter of choice. Earlier I mentioned a lot of rules that an RDMS can implement and enforce. Using a document database, such as MongoDB, those rules have to be managed at the application layer. Also, in MongoDB there is no such thing as stored procedures, so you cannot use that for enforcing rules.

Database Implementation

MICROSOFT SQL SERVER

Tables and Constraints

In the table below you will find the SQL script used to create the SenseiDB tables and constraints in Microsoft SQL Server.

```
-- =====
-- Author:          Brian Munksgaard
-- Creation date:    05-12-2015
-- Description:      This script is used to drop and
--                  create all the tables used in
--                  used in the SenseiDB.
-- =====
USE SenseiDB;

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

-----
-- Drop all tables.
-----
IF OBJECT_ID('dbo.Review', 'U') IS NOT NULL DROP TABLE dbo.Review;
IF OBJECT_ID('dbo.Attending', 'U') IS NOT NULL DROP TABLE dbo.Attending;
IF OBJECT_ID('dbo.CourseTagAssignment', 'U') IS NOT NULL DROP TABLE
dbo.CourseTagAssignment;
IF OBJECT_ID('dbo.CourseLocation', 'U') IS NOT NULL DROP TABLE dbo.CourseLocation;
IF OBJECT_ID('dbo.CourseTag', 'U') IS NOT NULL DROP TABLE dbo.CourseTag;
IF OBJECT_ID('dbo.CourseLesson', 'U') IS NOT NULL DROP TABLE dbo.CourseLesson;
IF OBJECT_ID('dbo.Location', 'U') IS NOT NULL DROP TABLE dbo.Location;
IF OBJECT_ID('dbo.City', 'U') IS NOT NULL DROP TABLE dbo.City;
IF OBJECT_ID('dbo.Course', 'U') IS NOT NULL DROP TABLE dbo.Course;
IF OBJECT_ID('dbo.CourseGroup', 'U') IS NOT NULL DROP TABLE dbo.CourseGroup;
IF OBJECT_ID('dbo.PersonCredentials', 'U') IS NOT NULL DROP TABLE
dbo.PersonCredentials;
IF OBJECT_ID('dbo.Person', 'U') IS NOT NULL DROP TABLE dbo.Person;

-----
-- Create all tables.
-----
CREATE TABLE dbo.Person (
    PersonUserName nvarchar(30) NOT NULL PRIMARY KEY,
    PersonFirstName nvarchar(30) NOT NULL,
    PersonLastName nvarchar(30) NOT NULL,
    PersonEmail nvarchar(75) NOT NULL,
    PersonBirthday date NOT NULL
) ON [PRIMARY];

CREATE TABLE dbo.PersonCredentials (
    UserName nvarchar(30) NOT NULL PRIMARY KEY,
    UserPass nvarchar(100) NOT NULL
) ON [PRIMARY];

CREATE TABLE dbo.CourseGroup (
    CourseGroupId int IDENTITY(1,1) PRIMARY KEY,
    CourseGroupName nvarchar(30) NOT NULL UNIQUE,
    CourseParentGroup int
) ON [PRIMARY];

CREATE TABLE dbo.Course (
    CourseId int IDENTITY(1,1) PRIMARY KEY,
    CourseTitle nvarchar(30) NOT NULL,
    CourseDescription nvarchar(1000) NOT NULL,
    CoursePrerequisites nvarchar(500) NOT NULL,
    CoursePriceHour int NOT NULL,
    CourseMaxAttendess int NOT NULL,
    CourseStartDate date NOT NULL,
```

```

        CourseEndDate date NOT NULL,
        CourseIsActive int NOT NULL CHECK(CourseIsActive = 0 OR CourseIsActive = 1),
        CourseGroupId int NOT NULL FOREIGN KEY REFERENCES
CourseGroup(CourseGroupId),
        TeacherUserName nvarchar(30) NOT NULL FOREIGN KEY REFERENCES
Person(PersonUserName)
) ON [PRIMARY];

CREATE TABLE dbo.City (
    CityZipCode nchar(10) NOT NULL,
    CityName nvarchar(30) NOT NULL
    PRIMARY KEY (CityZipCode)
) ON [PRIMARY];

CREATE TABLE dbo.Location (
    LocationId int IDENTITY(1,1) PRIMARY KEY,
    LocationName nvarchar(50) NOT NULL,
    LocationStreet nvarchar(50) NOT NULL,
    LocationStreetNumber int NOT NULL,
    LocationZipCode nchar(10) NOT NULL FOREIGN KEY REFERENCES City(CityZipCode)
) ON [PRIMARY];

CREATE TABLE dbo.CourseLesson (
    CourseId int FOREIGN KEY REFERENCES Course(CourseId),
    CourseDay int NOT NULL CHECK(CourseDay >= 1 AND CourseDay <= 7),
    CourseStartTime time NOT NULL,
    CourseEndTime time NOT NULL
    PRIMARY KEY (CourseId, CourseDay)
) ON [PRIMARY];

CREATE TABLE dbo.CourseTag (
    TagName nvarchar(30) PRIMARY KEY
) ON [PRIMARY];

CREATE TABLE dbo.CourseLocation (
    CourseId int FOREIGN KEY REFERENCES Course(CourseId),
    LocationId int FOREIGN KEY REFERENCES Location(LocationId)
    PRIMARY KEY (CourseId, LocationId)
) ON [PRIMARY];

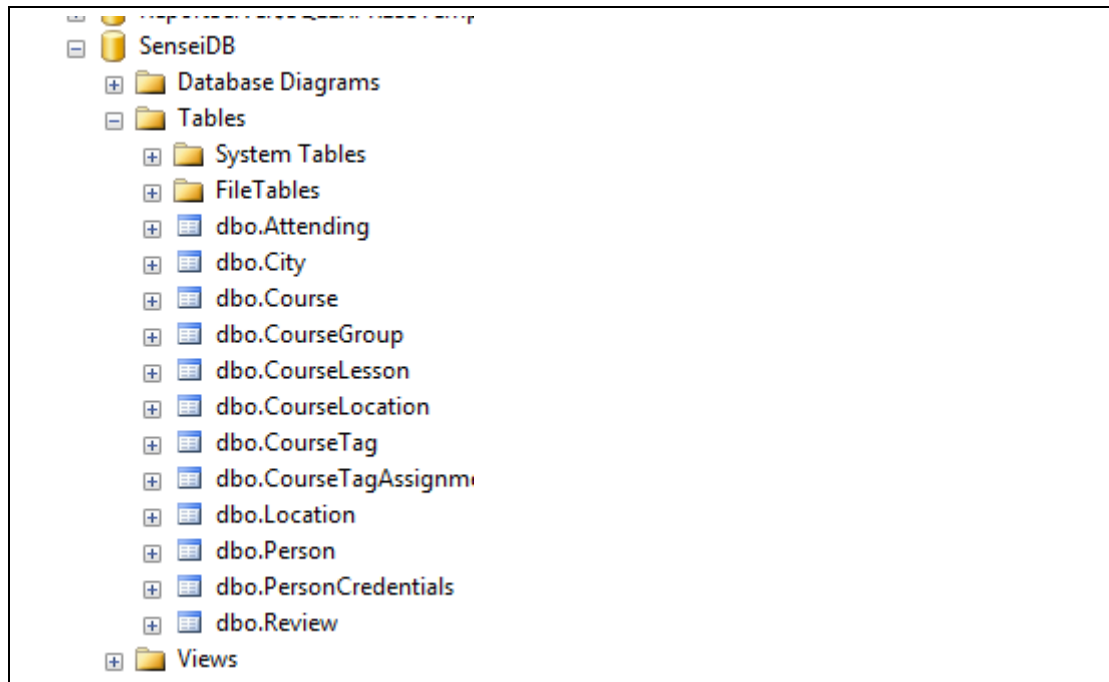
CREATE TABLE dbo.CourseTagAssignment (
    CourseId int FOREIGN KEY REFERENCES Course(CourseId),
    TagName nvarchar(30) FOREIGN KEY REFERENCES CourseTag(TagName)
    PRIMARY KEY (CourseId, TagName)
) ON [PRIMARY];

CREATE TABLE dbo.Attending (
    AttendingUserName nvarchar(30) FOREIGN KEY REFERENCES
Person(PersonUserName),
    AttendingCourseId int FOREIGN KEY REFERENCES Course(CourseId)
    PRIMARY KEY(AttendingUserName, AttendingCourseId)
) ON [PRIMARY];

CREATE TABLE dbo.Review (
    CourseId int FOREIGN KEY REFERENCES Course(CourseId),
    ReviewId int IDENTITY(1,1),
    ReviewDate datetime NOT NULL,
    ReviewRating int NOT NULL CHECK(ReviewRating >= 0 AND ReviewRating <= 5),
    ReviewText nvarchar(500) NOT NULL,
    ReviewUserName nvarchar(30) FOREIGN KEY REFERENCES Person(PersonUserName)
    PRIMARY KEY(CourseId, ReviewId)
) ON [PRIMARY];

```

And after running the script in SQL Server Management Studio we can see that the tables have been created:



Stored Procedures

Using stored procedures is a nice way to encapsulate multiple table updates and transactions. Following, I have described and created two stored procedures that are relevant in the SenseiDB, the CreateCourse and AssignTag stored procedures.

Create Course Stored Procedure

In the data model you can see that a course always need a location, hence the one-to-one and always one relations. Therefore, when creating a new course we need to insert data in three tables: Course, Location and CourseLocation. If one of the inserts fail, all fails (the ACID principles).

In the table below you can see the SQL code for the CreateCourse procedure:

```
USE SenseiDB;

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

IF OBJECT_ID('dbo.CreateCourse') IS NOT NULL DROP PROCEDURE dbo.CreateCourse;
GO

-- =====
-- Author:      Brian Munksgaard
-- Create date: 05-12-2015
-- Description: This stored procedure is used to
--              create a new course and the
--              associated location.
-- =====
CREATE PROCEDURE dbo.CreateCourse
    @CourseTitle nvarchar(30),
    @CourseDescription nvarchar(1000),
```

```

@CoursePrerequisites nvarchar(500),
@CoursePriceHour int,
@CourseMaxAttendess int,
@CourseStartDate date,
@CourseEndDate date,
@CourseIsActive int,
@CourseGroupId int,
@TeacherUserName nvarchar(30),
@LocationName nvarchar(50),
@LocationStreet nvarchar(50),
@LocationStreetNumber int,
@LocationZipCode nvarchar(10)
AS
BEGIN

    BEGIN TRANSACTION CreateCourseTransaction;

    INSERT INTO dbo.Course (CourseTitle, CourseDescription, CoursePrerequisites,
                           CoursePriceHour, CourseMaxAttendess,
                           CourseStartDate, CourseEndDate, CourseIsActive,
                           CourseGroupId, TeacherUserName)
    VALUES (@CourseTitle, @CourseDescription, @CoursePrerequisites,
            @CoursePriceHour, @CourseMaxAttendess,
            @CourseStartDate, @CourseEndDate, @CourseIsActive,
            @CourseGroupId, @TeacherUserName);

    DECLARE @CourseId AS int;
    SET @CourseId = SCOPE_IDENTITY();

    INSERT INTO dbo.Location (LocationName, LocationStreet,
                             LocationStreetNumber, LocationZipCode)
    VALUES (@LocationName, @LocationStreet,
            @LocationStreetNumber, @LocationZipCode);

    DECLARE @LocationId AS int;
    SET @LocationId = SCOPE_IDENTITY();

    INSERT INTO dbo.CourseLocation(CourseId, LocationId)
    VALUES (@CourseId, @LocationId);

    IF @@ERROR <> 0
        BEGIN
            ROLLBACK TRANSACTION CreateCourseTransaction;
            RETURN 1;
        END
    ELSE
        BEGIN
            COMMIT TRANSACTION CreateCourseTransaction;
            RETURN 0;
        END;

END;
GO

```

Assign Tag Stored Procedure

This stored procedure is also used to handle multiple transaction based inserts.

First, the procedure checks whether or not the tag is already assigned to the course. If so, the procedure just return 0 to indicate success. Second, the procedure checks whether or not the tag exists. If not, the tag is created. Finally, the tag is assigned to the course. And, of course, on errors everything is rolled back.

```

USE SenseiDB;

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

IF OBJECT_ID('dbo.AssignTag') IS NOT NULL DROP PROCEDURE dbo.AssignTag;
GO

```

```

-- =====
-- Author:      Brian Munksgaard
-- Create date: 05-12-2015
-- Description: This stored procedure is used to
--              assign a tag to a course. If the
--              tag does not exists, it will be
--              created.
-- =====
CREATE PROCEDURE dbo.AssignTag
    @CourseId int,
    @TagName nvarchar(30)
AS
BEGIN

    BEGIN TRANSACTION AssignTagTransaction;

    -- First, check whether or not the tag is
    -- already assigned to the course. If so
    -- just return SUCCESS.
    IF EXISTS
    (
        SELECT 1
        FROM    dbo.CourseTagAssignment a
        WHERE   a.CourseId = @CourseId
              AND a.TagName = @TagName
    )
    BEGIN
        RETURN 0;
    END;

    -- Second, check whether or not the tag exists.
    -- If not, create the tag and continue.
    IF NOT EXISTS
    (
        SELECT 1
        FROM    dbo.CourseTag A
        WHERE   a.TagName = @TagName
    )
    BEGIN
        INSERT INTO dbo.CourseTag (TagName)
        VALUES (@TagName)
    END;

    -- Finally, assign the tag to the course.
    INSERT INTO dbo.CourseTagAssignment (CourseId, TagName)
    VALUES (@CourseId, @TagName);

    -- Any errors, rollback everything.
    -- Otherwise commit all inserts.
    IF @@ERROR <> 0
    BEGIN
        ROLLBACK TRANSACTION AssignTagTransaction;
        RETURN 1;
    END
    ELSE
    BEGIN
        COMMIT TRANSACTION AssignTagTransaction;
        RETURN 0;
    END;
END;
GO

```

Using the Stored Procedures

Below you can see how the stored procedures are used. First I create a course:

```

EXEC dbo.CreateCourse
    @CourseTitle = 'Golf for begyndere og let øvede',
    @CourseDescription = 'Hej, jeg er en garvet golfspiller, som har lyst til at
lære andre, at spille golf. Så har jeg jo også nogen at spille med :-) Er det noget
for dig, er du meget velkommen til at kontakte mig.',
    @CoursePrerequisites = 'Ingen til at starte med, du låner udstyr af mig. Med
tiden skal du dog anskaffe dig dit eget.',
    @CoursePriceHour = 30,

```

```

@CourseMaxAttendess = 2,
@CourseStartDate = '20160107',
@CourseEndDate = '20160512',
@CourseIsActive = 1,
@CourseGroupId = 7,
@TeacherUserName = 'lm',
@LocationName = 'Ikast Golfklub',
@LocationStreet = 'Remmevej',
@LocationStreetNumber = 36,
@LocationZipCode = '7430'

```

And then I create and assign some tags to the course:

```

USE SenseiDB;

EXEC dbo.AssignTag
    @CourseId = 3,
    @TagName = 'MEGASWAGGOLFING'

EXEC dbo.AssignTag
    @CourseId = 3,
    @TagName = 'CRAP'

EXEC dbo.AssignTag
    @CourseId = 3,
    @TagName = 'NICE'

```

Views

Views are good for more complex, often used, join constructions. I have included two view examples that may apply to the SenseiDb, Top5GroupHierarchy and CourseOverview.

Top5GroupHierarchy

This view is used to list group hierarchy for the first five levels (from the top). The view gives you a quick overview of the group hierarchy and can also be used by an application, to create a graphical representation of the group hierarchy. Below you can see the code used to create the view:

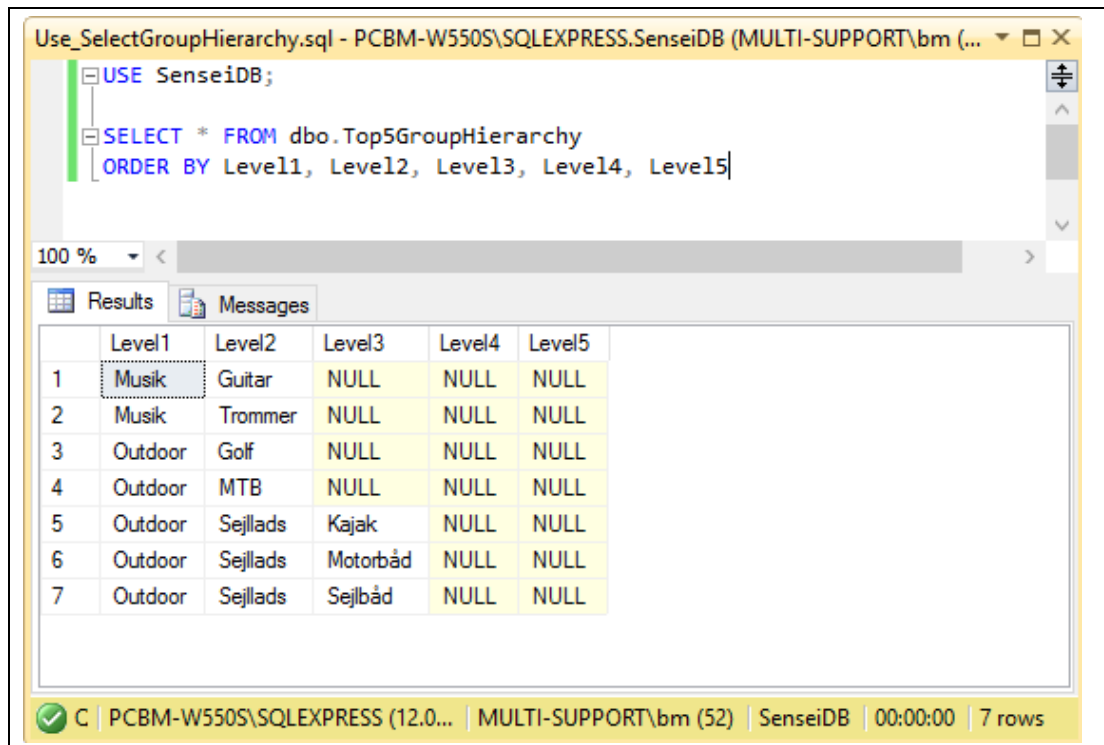
```

-- =====
-- Author:      Brian Munksgaard
-- Creation date: 06-12-2015
-- Description: This view is used to list
--              the hierarchy and the group
--              names of top 5 groups.
-- =====
USE SenseiDB;
GO

-- Retrieve groups hierachy and group name
-- for the five top level groups.
CREATE VIEW dbo.Top5GroupHierarchy
AS
SELECT a.CourseGroupName AS Level1,
       b.CourseGroupName AS Level2,
       c.CourseGroupName AS Level3,
       d.CourseGroupName AS Level4,
       e.CourseGroupName AS Level5
FROM   dbo.CourseGroup AS a
JOIN   dbo.CourseGroup AS b ON a.CourseGroupId = b.CourseParentGroup
LEFT OUTER JOIN  dbo.CourseGroup AS c ON b.CourseGroupId = c.CourseParentGroup
LEFT OUTER JOIN  dbo.CourseGroup AS d ON c.CourseGroupId = d.CourseParentGroup
LEFT OUTER JOIN  dbo.CourseGroup AS e ON d.CourseGroupId = e.CourseParentGroup
WHERE  a.CourseParentGroup = 0

```

And the view is used like this:



The screenshot shows a SQL query window titled 'Use_SelectGroupHierarchy.sql - PCBM-W550S\SQLEXPRESS, SenseiDB (MULTI-SUPPORT\bm (...))'. The query is as follows:

```
USE SenseiDB;
SELECT * FROM dbo.Top5GroupHierarchy
ORDER BY Level1, Level2, Level3, Level4, Level5
```

The 'Results' tab displays the following data:

	Level1	Level2	Level3	Level4	Level5
1	Musik	Guitar	NULL	NULL	NULL
2	Musik	Trommer	NULL	NULL	NULL
3	Outdoor	Golf	NULL	NULL	NULL
4	Outdoor	MTB	NULL	NULL	NULL
5	Outdoor	Sejllads	Kajak	NULL	NULL
6	Outdoor	Sejllads	Motorbåd	NULL	NULL
7	Outdoor	Sejllads	Sejlbåd	NULL	NULL

The status bar at the bottom indicates: 'PCBM-W550S\SQLEXPRESS (12.0...) | MULTI-SUPPORT\bm (52) | SenseiDB | 00:00:00 | 7 rows'.

CourseOverview

This view is used to retrieve a course overview with information such as location, city and teacher name. The view is good for course searches by ZipCode, TeacherName and/or GroupName. Only active courses are retrieved.

The view is created with the following SQL:

```
-- =====
-- Author:      Brian Munksgaard
-- Creation date: 06-12-2015
-- Description:  This view is used to retrieve
--               a course overview with
--               information such as location,
--               city and teacher name.
--               The view is good for course
--               searches by ZipCode,
--               TeacherName and/or GroupName.
-- =====
USE SenseiDB;
GO

DROP VIEW dbo.CourseOverview;
GO

CREATE VIEW dbo.CourseOverview
AS
SELECT a.CourseId, a.CourseTitle, c.LocationName, LocationStreet =
CONCAT(c.LocationStreet, ' ', c.LocationStreetNumber), d.CityZipCode, d.CityName,
TeacherName = CONCAT(e.PersonFirstName, ' ', e.PersonLastName), f.CourseGroupName
FROM dbo.Course AS a, dbo.CourseLocation b, dbo.Location c, dbo.City d, Person e,
CourseGroup f
WHERE a.CourseId = b.CourseId
AND b.LocationId = c.LocationId
AND c.LocationZipCode = d.CityZipCode
AND a.TeacherUserName = e.PersonUserName
```

```
AND a.CourseGroupId = f.CourseGroupId
AND a.CourseIsActive = 1
```

For testing, this view is OK, but as soon as larger amounts of data is loaded into the database, indexes are needed to support this search.

Below you can see some examples of using the view:

The screenshot shows a SQL Server Enterprise Manager window titled 'Use_CourseOverview...-SUPPORT\bm (56)'. The query editor contains the following SQL code:

```
USE SenseiDB;

-- All courses.
SELECT * FROM dbo.CourseOverview

-- All courses in 8600 Silkeborg.
SELECT * FROM dbo.CourseOverview
WHERE CityZipCode = '8600'

SELECT * FROM dbo.CourseOverview
WHERE CourseGroupName = 'Guitar'
```

Below the query editor, the 'Results' tab is active, displaying three tables of data. The first table shows results for the first query (all courses), the second for the second query (courses in 8600 Silkeborg), and the third for the third query (courses in the 'Guitar' group).

CourseId	CourseTitle	LocationName	LocationStreet	CityZipCode	CityName	TeacherName	CourseGroupName	
1	1	Guitarundervisning for begynde	Erhvervsakadem Århus, Basement	Sønderhøj 30	8260	Viby J	Emil Munksgaard	Guitar
2	2	Sejllads for begyndere og let	Silkeborg Sejlklub	Hattenæs 2	8600	Silkeborg	Brian Munksgaard	Sejlbåd
3	3	Golf for begyndere og let øved	Ikast Golfklub	Remmevej 36	7430	Ikast	Lucas Munksgaard	Golf

CourseId	CourseTitle	LocationName	LocationStreet	CityZipCode	CityName	TeacherName	CourseGroupName	
1	2	Sejllads for begyndere og let	Silkeborg Sejlklub	Hattenæs 2	8600	Silkeborg	Brian Munksgaard	Sejlbåd

CourseId	CourseTitle	LocationName	LocationStreet	CityZipCode	CityName	TeacherName	CourseGroupName	
1	1	Guitarundervisning for begynde	Erhvervsakadem Århus, Basement	Sønderhøj 30	8260	Viby J	Emil Munksgaard	Guitar

MONGODB

From the document database diagram, you can deduct the collections needed to implement the database. All loosely coupled (with white diamond next to it) entities becomes a collection.

This gives us the collections: City, CourseGroup and Person. From the model you can also see that everything springs from the Course entity. Course entities, and all tightly coupled entities, will be stored in the Course collection.

Group Hierarchy

In the relational database, you can join a table with itself, performing a so-called self-join. That way the group names and their level could be displayed with one query. In Mongo you cannot do such thing, at least not with the current data structure. To do something similar in Mongo, you need an application to query and traverse the database.

All the groups can easily be listed though:

```
> db.CourseGroup.find({}, {_id: 0})
{ "CourseGroupName" : "Musik", "CourseParentGroup" : "" }
{ "CourseGroupName" : "Outdoor", "CourseParentGroup" : "" }
{ "CourseGroupName" : "Guitar", "CourseParentGroup" : "Musik" }
{ "CourseGroupName" : "Trommer", "CourseParentGroup" : "Musik" }
{ "CourseGroupName" : "Sejllads", "CourseParentGroup" : "Outdoor" }
{ "CourseGroupName" : "MTB", "CourseParentGroup" : "Outdoor" }
{ "CourseGroupName" : "Golf", "CourseParentGroup" : "Outdoor" }
{ "CourseGroupName" : "Kajak", "CourseParentGroup" : "Sejllads" }
{ "CourseGroupName" : "Sejlbåd", "CourseParentGroup" : "Sejllads" }
{ "CourseGroupName" : "Motorbåd", "CourseParentGroup" : "Sejllads" }
>
```

CourseOverview

Remember the CourseOverview view, used in Microsoft SQL Server RDMS. Mongo does not have Views, but with the Mongo find command we can do almost the same thing.

Find all courses, retrieve only fields we are interested in:

```
> db.Course.find({}, {CourseTitle: 1, Location: 1, TeacherUserName: 1,
CourseGroupName: 1}).pretty()
{
  "_id" : ObjectId("56661290a85f3da54804dae7"),
  "CourseTitle" : "Guitarundervisning for begyndere",
  "CourseGroupName" : "Guitar",
  "TeacherUserName" : "em",
  "Location" : {
    "LocationName" : "Erhvervsakademi Århus, Basement",
    "LocationStreet" : "Sønderhøj",
    "LocationStreetNumber" : 30,
    "LocationZipCode" : "8260"
  }
}
{
  "_id" : ObjectId("56661291a85f3da54804dae8"),
  "CourseTitle" : "Sejllads for begyndere og let øvede",
  "CourseGroupName" : "Sejlbåd",
```

```

    "TeacherUserName" : "bm",
    "Location" : {
      "LocationName" : "Silkeborg Sejlklub",
      "LocationStreet" : "Hattenæs",
      "LocationStreetNumber" : 2,
      "LocationZipCode" : "8600"
    }
  }
}
{
  "_id" : ObjectId("56661291a85f3da54804dae9"),
  "CourseTitle" : "Golf for begyndere og let øvede",
  "CourseGroupName" : "Golf",
  "TeacherUserName" : "lm",
  "Location" : {
    "LocationName" : "Ikast Golfklub",
    "LocationStreet" : "Remmevej",
    "LocationStreetNumber" : 36,
    "LocationZipCode" : "7430"
  }
}
}
>

```

Find all courses taking place in 8600 Silkeborg, retrieve only fields we are interested in:

```

> db.Course.find({'Location.LocationZipCode': "8600"}, {CourseTitle: 1, Location:
1, TeacherUserName: 1, CourseGroupName: 1}).pretty()
{
  "_id" : ObjectId("56661291a85f3da54804dae8"),
  "CourseTitle" : "Sejllads for begyndere og let øvede",
  "CourseGroupName" : "Sejlbåd",
  "TeacherUserName" : "bm",
  "Location" : {
    "LocationName" : "Silkeborg Sejlklub",
    "LocationStreet" : "Hattenæs",
    "LocationStreetNumber" : 2,
    "LocationZipCode" : "8600"
  }
}
}
>

```

Find all guitar courses, retrieve only fields we are interested in:

```

> db.Course.find({CourseGroupName: "Guitar"}, {CourseTitle: 1, Location: 1,
TeacherUserName: 1, CourseGroupName: 1}).pretty()
{
  "_id" : ObjectId("56661290a85f3da54804dae7"),
  "CourseTitle" : "Guitarundervisning for begyndere",
  "CourseGroupName" : "Guitar",
  "TeacherUserName" : "em",
  "Location" : {
    "LocationName" : "Erhversakademi Århus, Basement",
    "LocationStreet" : "Sønderhøj",
    "LocationStreetNumber" : 30,

```

```
        "LocationZipCode" : "8260"  
    }  
}  
>
```

Late changes

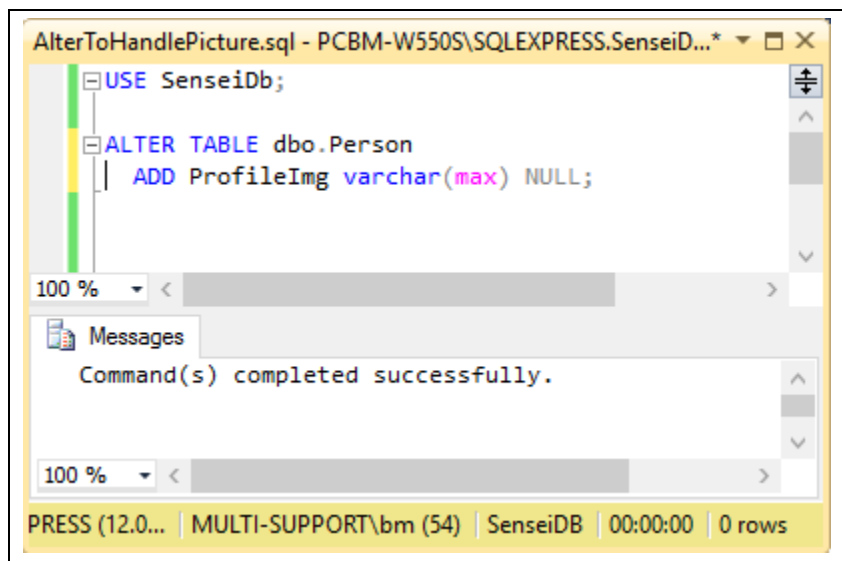
MISSING ATTRIBUTES

Going through the report I discovered that I have missed the profile picture from profile page. However, the changes needed in order to assign a picture to a person entity, can be implemented rather easily.

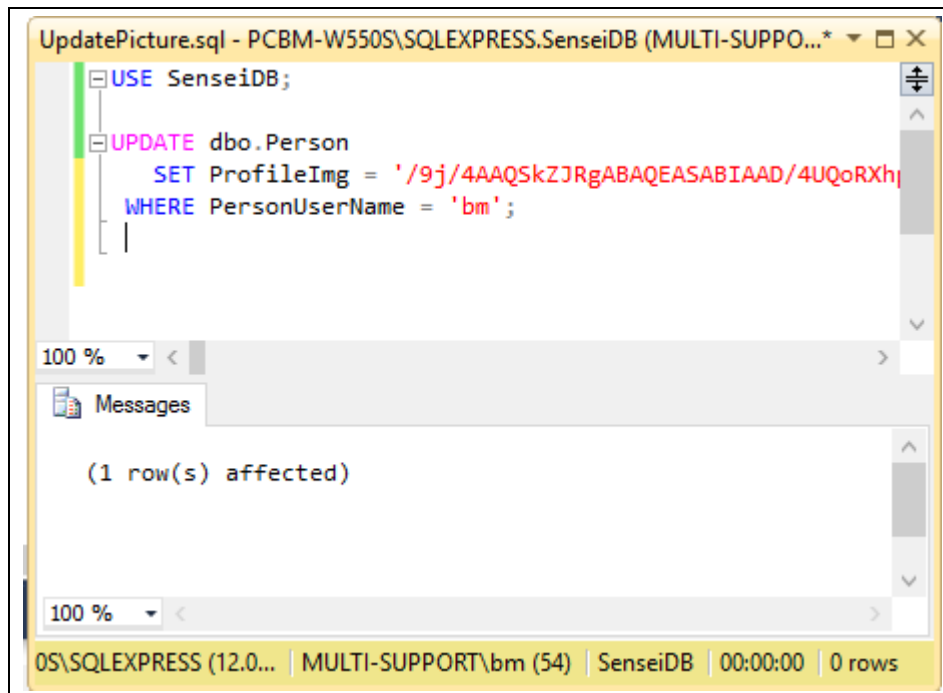
Assuming that pictures are stored as base64 encoded string we can store the data as shown below.

Relational Database Changes

In the RDBMS I would create an extra attribute on the Person table using the ALTER TABLE command:



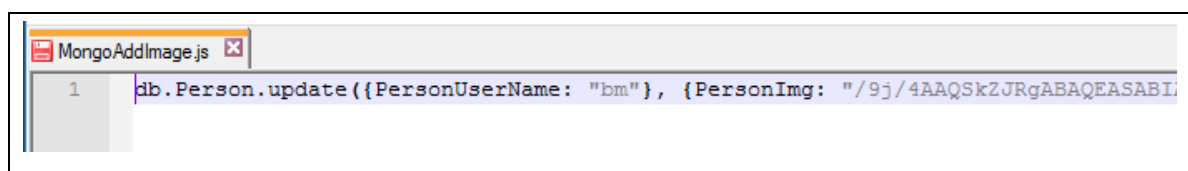
And then update the profile image on user 'bm':



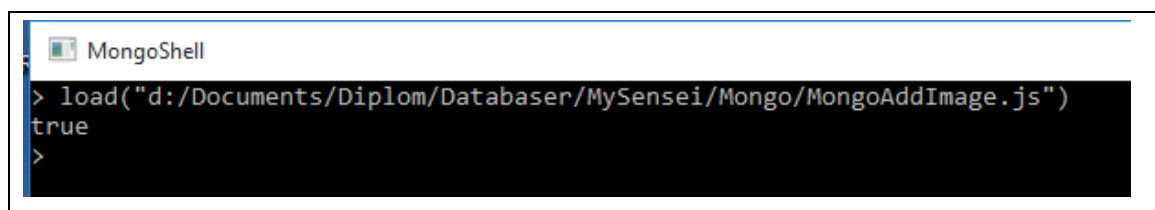
MongoDB Changes

In Mongo nothing needs to be changed. As you might recall, MongoDB is a schema-less database. In order to add a profile image to a person you just update the document.

Below you can see part of a MongoDB script that will add an image to the user/person 'bm':



From the Mongo shell I load and execute the script:



And querying the user 'bm' you can see that the data has been added:

[illegible]

Final Notes

CHOOSING THE DATABASE

So, what database should you choose? Well there is no easy answer to that question. Basically, it all comes down to the application(s) using the database, how data is accessed, how many inserts, how many queries, the type data being stored, what statistics will be extracted. Other things to think about is the database itself, scaling, backup, how well known is the DB, support, programming languages etc. Do you even have a choice? Often the database system is already given and cannot be switched.

In this scenario, I started out with a relational database. The dataset is very much relational and easily normalized to the third normal form. However, migrating the data to a document database was also a rather easy process, which actually surprised me. In my opinion, both are valid choices.

For the MySensei service I tend to lean toward the relational database system and in this case, Microsoft SQL Server.

My reasons for this are:

- It is a well-known and well-proven RDBMS.
- It is widely used.

- It is known to many database administrators and application programmers, making it easier to find the right people to develop and maintain the DB and applications.
- Same supplier/developer (Microsoft) for the DB, the IDE (Integrated Development Environment) and the programming languages and frameworks.
- I assume, it will be easier to find a hosted SQL server solution than a hosted MongoDB.

As the service grows and more data is stored, the choice of database may have to be reconsidered. May you need to switch to another database or maybe you need to use both a relational database and a document database. Again, it all depends on the data and what you want to do with it.