Auch hier nutze ich Docker um eine MongoDB Instanz zu starten. Anschließend öffne ich über das Docker Dashboard wieder das Terminal des Containers.

```
docker run --name mongo-test -d mongo
```

In dem Terminal des Containers geben wir `mongo` ein um die MongoDB Shell zu öffnen.

```
mongo
```

MongoDB nutzt sogenannte `Dokumente` um Daten abzuspeichern. Sie sind vergleichbar mit den Zeilen in einer Relationalen Datenbank, jedoch mit dem Unterschied das sie nicht in ein Schema gezwungen werden, wie es bei Relationalen Datenbanken durch die Spalten der Fall ist. Das heißt ein Dokument ist quasi wie ein `JSON`, `YAML` oder `XML` Objekt, das in der Struktur komplett frei wählbar ist. Ein Dokument ist also eine Sammlung an `Key-Value` Paaren, wobei der Wert eines solchen Paares alles mögliche annehmen kann. Man kann z.B. `BLOB`s (Binäre Daten), Daten in allen möglichen Dateiformaten, aber auch auch weitere `Key-Value` Paare in den Wert eines `Key-Value` Paar speichern. Das erlaubt es z.B. verschachtelte Objekte zu erschaffen und bietet eine sehr große Flexibilität.

Mehrere Dokumente können in eine `Collection` zusammengefasst werden. Dies entspricht dann quasi der Tabelle, welche mehrere Zeilen (hier `Dokumente`) enthält.

Als erstes schauen wir uns das alt bekannte Beispiel der Musik Playlist an. Wir haben wieder mehrere Songs, die zu einer Sammlung zusammengefasst werden können. Jeder Song hat wieder einen `SongName` ein `year` und einen `singer`. Das erstellen eines Primary Keys ist hier nicht notwendig, da MongoDB jedem Dokument eine eindeutige ID zuweist. Diese findet man später als `_id` im Dokument wieder.

Zunächst erstellen wir unsere Collection

```
db.createCollection("Music_Playlist")
```

Dann speichern wir uns die Collection in eine Variable, sodass wir statt `db.Music_Playlist` einfach `coll` nutzen können. Wir können uns fast alles in Variablen schreiben. Hier sieht man sehr gut, das die Mongo Shell sehr ähnlich einer NodeJS shell ist.

```
coll = db.Music_Playlist
```

Anschließend fügen wir ein Dokument in unsere Collection ein. Das geht mit `coll.insertOne(<Dokument>)`. Hierbei nutze wir das `JSON` Format.

```
coll.insertOne(
  { SongName: 'Song 1 Name', year: 2020, singer: 'Singer 1 Name' }
```

```
)
```

Jetzt können wir mit der `find()` Methode uns alle Dokumente aus der Collection holen.

```
coll.find()
```

Mit `insertMany(<[Dokumente]>)` können wir auch mehrere Dokumente anlegen. Dazu übergeben wir der Funktion ein Array aus Dokumenten.

```
coll.insertMany(
  [
    { SongName: 'Song 2 Name', year: 2020, singer: 'Singer 2 Name' },
    { SongName: 'Song 3 Name', year: 2015, singer: 'Singer 3 Name' },
    { SongName: 'Song 4 Name', year: 2010, singer: 'Singer 4 Name' },
    { SongName: 'Song 5 Name', year: 2000, singer: 'Singer 5 Name' },
  ]
)
```

Der erste Parameter der `find()` Methode erlaubt es uns die Dokumente zu filtern. Ähnlich dem `WHERE` Keyword aus SQL. Wir übergeben ein Object an die `find()` Methode, das ein oder mehrere Key-Value Paare hat. Das Beispiel unten wäre äquivalent zu `WHERE year=2020`. Wenn wir zwei Key-Value Paare übergeben würden. Dann nimmt MongoDB implizit ein `AND` an. Das heißt `coll.find({year: 2020, singer: 'Singer 2'})` wäre equivalent zu: `WHERE year=2020 AND singer='Singer 2'`.

Für andere Logische Verknüpfungen und weitere Operationen wie `Not Equal` etc... gibt es in MongoDB die `Query Operators`.

Beispiel für alle Jahre die größer als 2010 sein sollen: `coll.find({ year: { $gt: 2010 } })`

```
coll.find({year: 2020 })
```

Um ein Update auf eine bereits existierendes Dokument auszuführen, können wir einfach die `upateOne` Methode nutzen. Diese Methode erwartet 2 Parameter:

1. Eine Query, nachdem wir das Dokument suchen (ist identisch mit dem 1. Parameter der `find` Methode.)
2. Werte die wir updaten wollen.

Beim 2. Parameter können wir mit 2 Methoden des Updates arbeiten. Wenn wir einfach ein JSON Objekt übergeben, dann wird dieses JSON Objekt in das Dokument geschrieben und alle vorherigen Daten, die nicht in dem übergebenen JSON Objekt sind werden gelöscht. Das ganze ist also äquivalent zum löschen des Dokuments und anschließendem erstellen des Dokuments mit gleicher `_id` und den Daten aus dem 2. Parameter der `updateOne()` Methode.

Die zweite Option ist wie unten über den `$set` Operator zu gehen. Dabei werden die beiden Objekte zusammengefasst und alle Werte des Dokuments werden mit Werten aus dem Parameter Objekt

überschrieben. Wenn das Dokument einen Key enthält, der nicht in dem `$set` Objekt ist, bleibt der Key im Dokument unverändert.

```
coll.updateOne(
  { SongName: 'Song 1 Name' },
  { $set: {
    year: 1900
  }}
)

coll.find( { SongName: 'Song 1 Name' })
```

Relations:

Relationen können in MongoDB über zwei Methoden erreicht werden.

Embedding Pattern

Die erste Methode ist das `Embedding` von Sub-Dokumenten in einem Dokument.
Nehmen wir das vorherige Beispiel des Studenten und der Kurse. Wenn wir sagen, jeder Student kann nur einen Kurs belegen, also eine `1-1` Beziehung haben, dann können wir das ganz einfach die Daten des Kurses in das Dokument des Studenten einbetten. Beispiel wäre unten der Student mit dem `name Student Name` besucht den Kurs mit dem Name `Course 1 Name`. Dieses Pattern funktioniert für `1-1` bzw auch für `1` Kurs zu `n` Studierende, da wir mehrere Dokumente für die Studierenden anlegen können und dann im `course` Key die gleichen Daten in verschiedenen Studierende eintragen können.

```
student: {
  _id: <ObjectID1>
  name: 'Student Name',
  // Subdoc
  course: {
    name: 'Course 1 name'
  }
}
```

Vorteil:

- Alle Daten können ohne `JOIN` erreicht werden und es wird nur eine einzige Abfrage benötigt.

Nachteile:

- Kann zu sehr großen Dokumenten führen, die vlt nicht so häufig genutze Daten jedes mal mitladen müssen.

Subset Pattern

Der zweite Ansatz entspricht eher eine Relationalen Datenbank. Hierbei legen wir für jeden Studenten und jeden Kurs ein eigenes Dokument and und verknüpfen diese beiden dann anschließend zusammen indem wir in mindesten einen Dokument die `_id` des anderen Dokuments schreiben.

```
course: {
  _id: <ObjectIDCourse>,
  name: 'Course 1 name',
}

student: {
  _id: <ObjectIDStudent>,
  name: 'Student Name',
  course_id: <ObjectIDCourse>
}
```

Vorteil:

- Keine / Weniger Duplikation
- Nicht häufig genutzte Daten werden nicht jedes mal mitgeladen.

Nachteil:

- Es werden mehr als eine Abfrage benötigt um alle Daten zu holen, da es mehrere Dokumente sind.

Subset Beispiel

Hier ist einmal ein Beispiel, wie wir mit den Subset Pattern eine 1-1 Beziehung modellieren können. In dem Beispiel zwischen Studierenden und Kursen.

Wir erstellen zunächst zwei Kollektionen.

```
db.createCollection('Students')
db.createCollection('courses')

students = db.students
courses = db.courses
```

Jetzt legen wir uns einen Studierenden und einen Kurs an.

```
students.insertOne(
  {
    _id: 1,
    name: 'student 1',
  }
)

courses.insertOne(
  {
```

```
    _id: 1,
    name: 'Course 1',
  }
)
```

Jetzt verknüpfen wir das Kurs Dokument mit dem Studierenden Dokument

```
students.updateOne(
  { _id: 1 },
  { $set: {
    course_id: 1
  }}
)
```

Und lassen uns beides anzeigen.

```
students.find()
courses.find()
```

Mit `aggregate` können wir jetzt eine `JOIN` Operation ausführen. Dazu sagen wir über den `$lookup` Operator, wie die `JOIN` aussehen soll.

In dem Fall sagen wir, dass wir alle studierenden haben wollen und die mit der `courses` Kollektion zusammenfügen wollen. MongoDB soll das Feld `course_id` aus den Studierenden mit dem `_id` Feld der Courses zusammenführen und das ganze dann als `course` in dem Studierenden einbetten.

```
students.aggregate([
  {
    $lookup: {
      from: 'courses',
      localField: 'course_id',
      foreignField: '_id',
      as: 'course'
    }
  }
])
```

One-to-many Beispiel

Jetzt mal ein Beispiel für eine 1-m Beziehung.
Wir fügen zunächst einen weiteren Kurs ein.

```
courses.insertOne(
  {
    _id: 2,
    name: 'Course 2',
  }
)

courses.find()
```

Um dann anschließend diesen neuen Kurs unserem bereits vorhandenen Studierenden zuzufügen. Dazu schreiben wir jetzt ein Key `course_ids`, das wir dann mit einem Array aus `_id`s füllen. Hier in dem Fall `1` und `2`

```
// Add relation to student
students.updateOne(
  { _id: 1 },
  { $set: {
    course_ids: [1, 2]
  }}
)
```

Jetzt können wir wieder über `aggregate` einen `JOIN` ausführen. Dieser sieht identisch zum oberen aus, mit dem Unterschied, dass wir jetzt das Alias `courses` verwenden und das `localField` ist `course_ids`.

```
students.aggregate([
  {
    $lookup: {
      from: 'courses',
      localField: 'course_ids',
      foreignField: '_id',
      as: 'courses'
    }
  }
])
```

Many-to-many

Eine m-n Beziehung können wir sehr ähnlich einer 1-n Beziehung bauen. Nur das wir jetzt in beide Dokumente die jeweilig anderen `_id`s schreiben. Also in ein `course` Dokument schreiben wir alle dazugehörigen `students` und in ein `student` schreiben wir all sein `courses`.

```
courses: [
  {
    _id: 1,
    name: 'course 1',
  },
```

```
    {
      _id: 2,
      name: 'course 2',
      student_ids: [1, 2],
    }
  ]

  students: [
    {
      _id: 1,
      name: 'student 1',
      course_ids: [1, 2],
    }
    {
      _id: 2,
      name: 'student 2',
      course_ids: [1],
    }
  ]
```

Man kann diese Beziehung auch nur von einer Seite modellieren, indem man quasi eine 1-n Beziehung baut und z.B. nur die `course_ids` in die Studierenden schreibt und hierbei dann auch erlaubt, dass eine `_id` in mehreren `course_ids` Arrays stehen darf. Der Nachteil dabei ist, dass die Beziehung dann nur von einer Seite aus erreichbar ist.