Im vorigen Arbeitsblatt haben wir uns hauptsächlich damit beschäftigt unseren Server für MongoDB einzurichten und erste Schritte mit Mongoose zu unternehmen. In diesem Arbeitsblatt gibt es mehr Details darüber wie mit Mongoose gearbeitet werden kann.

Sehen wir uns zunächst an, wie wir ein besseres Schema mit Mongoose definieren können.

Länge der Datenfelder

Im Gegensatz zu relationalen Datenbanken ist es nicht nötig, die Länge der Datenfelder einzustellen. Das kann nützlich sein, wenn du die Menge der Daten, die in einem bestimmten Objekt gespeichert werden, ändern musst. Zum Beispiel könnte dein System ein 16-Zeichen-Limit für Benutzernamen vorschreiben. Später stellst du vielleicht fest, dass du die Benutzernamen verschlüsseln möchtest, aber das würde die Länge der gespeicherten Daten verdoppeln. Wenn dein Datenbankschema feste Feldgrößen verwendet, musst du es umgestalten, was bei einer großen Datenbank sehr lange dauern kann. Mit Mongoose kannst du einfach anfangen, das Datenobjekt zu verschlüsseln, ohne dich darum zu kümmern.

Datentypen

Die wichtigsten Typen sind:

- String UTF-8
- Number Achtung: Mongoose unterstützt die Datentypen long und double, MongoDB hingegen schon
- Date Datum und Zeit Objekt, typischerweise zurückgegeben von MongoDB als ISODate Objekt
- Boolean Boolsche Werte
- Buffer Binäre Informationen, zum Beispiel Bilder, die in MongoDB gespeichert sind
- ObjectId Fremdschlüssel zum Referenzieren eines anderen Dokuments
- Array Alle Werte müssen vom selben Typ sein

Validators

Alternativ kannst du den Typ auch über ein Objekt definieren:

```
const dog = new Schema({
  name: { type: String },
  born: { type: Date },
});
```

Nun kannst du noch weitere Properties (**Pathes** in Mongoose genannt) hinzufügen und weitere Checks[1] definieren.

```
const dog = new Schema({
  name: { type: String, default: 'wau wau', unique: true },
  born: { type: Date, min: new Date('2000-01-01'), max: Date.now, required: true },
  favFoods: [
    {
      type: String,
      minLength: 3,
      maxLength: 20,
    },
  ],
  sex: { type: String, enum: ['male', 'female'] },
  image: String,
});
```

[1] Siehe: https://mongoosejs.com/docs/validation.html

Diese Checks werden beim Einfügen in die Datenbank automatisch durchgeführt. Das ist ein weiterer Vorteil von Mongoose. Nehmen wir mal folgende Daten an:

```
[
  {
    "born": "1999-10-16",
    "name": "Cupcake",
    "favFoods": ["le", "meat"],
    "sex": "mehl",
    "image": "images/1.jpg"
  },
  {
    "born": "2001-04-22",
    "name": "Britney Ears",
    "favFoods": ["turnips", "corn"],
    "sex": "female",
    "image": "images/2.jpg"
  }
]
```

Diese verletzen die Regeln für `born`, `favFoods` und `sex`.

1. Gib die Fehlermeldungen übersichtlich aus. In jeder Zeile eine.

Lies dir dazu den Abschnitt über die Struktur der Error Meldung von Mongoose durch siehe: (https://mongoosejs.com/docs/validation.html)

2. Werden alle Objekte, die eingefügt werden sollen, gecheckt?

```
> node loadData.js

Mongoose connected
Error ====> Path `born` (Sat Oct 16 1999 01:00:00 GMT+0100 (GMT+01:00)) is before minimum allowed value (Sat Jan 01 2000
Error ====> `mehl` is not a valid enum value for path `sex`.
Error ====> Path `favFoods.0` (`le`) is shorter than the minimum allowed length (3).
Mongoose disconnected
Mongoose terminated through app termination
```

3. Lösche die alten Objekte und füge nun alle Objekte der Datei **dogs_extended.json** ein!

Abfragen: Query-By-Example

Die gängigsten Suchabfragen lassen sich über die `find` Methode auf einer Collection durchführen: Eine Variante dieses Suchens hast du schon im vorigen Arbeitsblatt kennengelernt. Die `find` Methode befindet sich, sowie alle anderen Methoden in Mongoose in dem jeweiligen Model Objekt, das aus dem Schema entstanden ist. Also in unserem Fall `Dog`.

Nun möchten wir uns das Ganze etwas genauer ansehen. Dabei kommt das Prinzip *Query-by-Example* zum Einsatz, bei dem die Suchkriterien im Wesentlichen mit einem Objekt (Dokument) beschrieben werden, das als Filter wirkt.

Es stehen hier die gewohnten logischen And-, Or- und Not-Verknüpfungen zur Verfügung und auch Vergleichsoperatoren (>, >=, =, <, <=), die durch spezielle Feldnamen, die mit einem **$** beginnen, umgesetzt sind.

Beispiel: Finde alle Hunde mit dem angegebenen Geschlecht:

```
Dog.find({ sex: { $eq: 'female' } });
```

4. Implementiere die Route und die dazu passende Datenbankabfrage, um alle Hunde mit einem angegebenen Geschlecht auszugeben.

```
GET http://localhost:3000/dogs?gender=male
```


```
[
  {
    "name": "Mr. Wow",
    "favFoods": ["lentils", "meat"],
    "_id": "60bcf90732f6693be4211143",
    "born": "2009-10-16T00:00:00.000Z",
    "sex": "male",
    "image": "images/1.jpg",
    "__v": 0
  },
  {
    "name": "Bark Twain",
    "favFoods": ["calf bones", "beef lung", "apples"],
    "_id": "60bcf90732f6693be4211145",
    "born": "2016-10-22T00:00:00.000Z",
    "sex": "male",
    "image": "images/3.jpg",
    "__v": 0
  }
  . . .
]
```


5. Implementiere die Route und die dazu passende Datenbankabfrage, um alle Hunde auszugeben (**`name`** und **`born`**), die seit einem angegebenen Datum geboren wurden!

```
GET http://localhost:3000/dogs?datebornsince='2012-12-12'
```


```
[
  {
    "name": "Bark Twain",
    "born": "2016-10-22T00:00:00.000Z"
  },
  {
    "name": "Cupcake",
    "born": "2017-03-12T00:00:00.000Z"
  },
  {
    "name": "Hector",
    "born": "2017-06-06T00:00:00.000Z"
  },
  . . .
]
```


Wie sieht es aus, wenn du alle Hunde finden willst, die **`apples`** als Lieblingsessen haben? In diesem Fall muss das Array **`favFoods`** durchsucht werden.

Wiederum ist das sehr einfach:

```
Dog.find({ favFoods: 'apples' });
```

6. Implementiere die Route und die dazu passende Datenbankabfrage, um alle Hunde auszugeben (**`name`** und **`favFood`**), bei denen das angegebene Futter in den **favFood** Array enthalten ist.

```
GET http://localhost:3000/dogs?favFood=corn
```

```
[
  {
    "name": "Britney Ears",
    "favFoods": ["turnips", "corn"]
  },
  {
    "name": "Lady Sarah Jessica Barker",
    "favFoods": ["calf bones", "corn"]
  },
  {
    "name": "Subwoofer",
    "favFoods": ["carrots", "bananas", "corn"]
  }
]
```

Wie sucht man nach mehreren Eigenschaften?

```
Dog.find({ $and: [{ favFoods: 'apples' }, { favFoods: 'beans' }] });
```

Übersichtlichere Version mit den neuen Helperfunktionen:

```
Dog.find().and([{ favFoods: 'apples' }, { favFoods: 'beans' }]);
```

Löschen

Es gibt hier mehrere Varianten, die sich durch den Namen selbst erklären[2]. Beispiele:

```
Dog.deleteOne({ name: { $eq: 'Sherlock Bones' } });
Dog.deleteMany({ sex: { $eq: 'female' } });
```

Oft hast du aber die **`id`** mittels **`req.body`** bekommen. Dann wäre eine andere Möglichkeit:

```
Dog.findByIdAndRemove(id);
```

7. Implementiere die Route und die dazu passende Datenbankabfrage, um einen Hund zu löschen. Wähle selbst, welche Variante du verwenden möchtest!

[2] Siehe: https://mongoosejs.com/docs/api/model.html

Update

Ähnlich wie beim Löschen. Siehe API-Dokumentation!

Beispiel:

```
Dog.updateOne({ name: nameOfDog }, { $set: { image: pic } });
```

8. Implementiere die Route und die dazu passende Datenbankabfrage, um Properties eines Hundes zu ändern. Wähle selbst, welche Variante du verwenden möchtest!