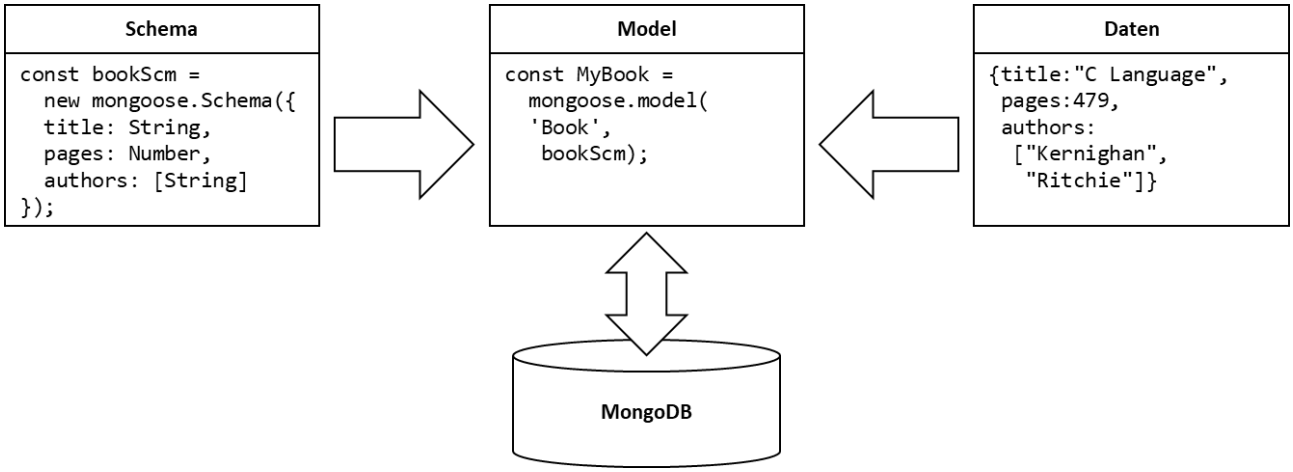
1 Grundidee von Mongoose

Grundsätzlich kann man direkt mit der MongoDB über die einzelnen Befehle, wie bspw. „`find()`“ zugreifen. Hierzu müssten wir lediglich den mongodb Treiber mittels `npm install express mongodb` installieren und könnten damit die Verbindung aufbauen und Daten austauschen. Mitunter möchte man jedoch die Zugriffslogik vereinheitlichen und die Kommunikation mit der MongoDB vereinfachen. Hier kommt Mongoose ins Spiel. Mongoose ermöglicht es uns, die Datenstrukturen in der MongoDB mittels eines Schemas festzulegen und daraus Modelle zu generieren. Diese wiederum sind JavaScript Objekte und synchronisieren sich mit der Datenbank – es findet also ein Mapping zwischen dem Model und der Datenbank statt.



Damit wir Mongoose nutzen können, müssen wir es über `npm` installieren:

```
npm install mongoose --save
```

Nun können wir mit Mongoose arbeiten. Hierzu müssen wir zum einen `mongoose` importieren und zum anderen die Datenbankverbindung aufbauen:

```
const mongoose = require("mongoose");
mongoose.connect("mongodb://localhost/myTestDb");
```

Wer noch zusätzliche Informationen für die Verbindung, wie bspw. Port, User oder Passwort benötigt, kann diese direkt in den Connection-String einbauen:

```
mongoose.connect("mongodb://myUser:myPwd@localhost:27017/myTestDb");
```

2 Schema

Ein Schema stellt eine strukturelle Anforderung an einen MongoDB Datensatz dar. Mongoose bietet hierfür das Objekt `Schema` an, welches im Konstruktor die Schemainformationen aufnimmt:

```
const BookSchema = new mongoose.Schema( {
    title: {type: String},
    pages: {type: Number},
    authors: [{type: String}]
});
```

Werden nur die Datentypen angegeben, so kann auch anstatt `{type: String}` nur `String` eingetragen werden.

Hier die wichtigsten Datentypen:

Datentyp:	Nutzung:	Beispiel Schema:	Beispiel Daten:
`String`	Zeichenketten	`type: String`	`"C Language"`
`Number`	Zahlen (mit/ohne Komma)	`type: Number`	`123.456`
`Date`	Datum	`type: Date`	`"2022-10-17T19:20:34"`
`Boolean`	Boolean Wert	`type: Boolean`	`True`
`Mixed`	Beliebige Datentypen	`type: {}`	`123.456`
`ObjectId`	Referenz auf eine MongoDB ID	`type: mongoose.ObjectId`	`otherBook._id`
`Array`	Arrays	`type: {String}`	`["A","B","C","D"]`
`Map`	Wie in JavaScript „Map“	`type: Map, of: String`	`{myKey: "myValue"}`
`Schema`	Anderes Schema	`type: yourSchema`	`{name: "peter", age: 21}`

Neben dem Datentyp können noch weitere Eigenschaften hinterlegt werden. Hier die gebräuchlichsten:

Eigenschaft:	Bedeutung:	Beispiel:
Die folgenden Eigenschaften gelten für alle Datentypen:		
`required`	True, wenn Eigenschaft vorhanden sein muss	`required: true`
`default`	Defaultwert, wenn Eigenschaft nicht belegt ist. Kann auch eine Funktion sein.	`default:"Mustermann"`
`validate`	Referenz auf Validierungsfunktion und Fehlermeldung	`validate: [validateTitle, 'title not valid']`
`get`	Individueller Getter für Zugriff auf Wert	`get:v=>myGetter(v)`
`index`	Index wird auf die Eigenschaft generiert	`index: true`
`unique`	Unique Index wird generiert	`unique: true`
Die folgenden Eigenschaften gelten für Strings		
`lowercase`	Beim Auslesen wird .toLowerCase() genutzt	`lowercase: true`
`uppercase`	Beim Auslesen wird .toUpperCase() genutzt	`uppercase: true`
`trim`	Beim Auslesen wird .trim() genutzt	`trim: true`
`match`	Wendet einen regulären Ausdruck für die Validierung an	`match:[/(\d{1,3}\.){2}\d{1,3}/, "no IP"]`
`enum`	Werte dürfen nur aus dem Array stammen	`enum:["A", "B"]`
`minLength`	Minimale Länge des Strings	`minLength: 10`
`maxLength`	Maximale Länge des Strings	`maxLength: 20`
Die folgenden Eigenschaften gelten für Number:		
`min`	Kleinster gültiger Wert	`min:-128`
`max`	Größter gültiger Wert	`max:127`

3 Model

Aus dem Schema wird nun ein Model „kompiliert“. Dies bedeutet, dass in JavaScript ein Objekt erzeugt wird, welches die Daten entsprechend dem Schema aufnehmen kann:

```
const BookModel = mongoose.model("Book", BookSchema);
```

Hier werden zwei Dinge festgelegt. Erstens, auf welchem Schema das Model basieren werden soll – in unserem Fall „`BookSchema`“. Weiterhin wird dem Model eine Collection zugewiesen, welche sich auf dem 1. Parameter „`Book`“ stützt und einfach die Mehrzahl darstellt – also „books“. Sollte ein fest vorgegebenes Schema benötigt werden, so muss ein dritter Parameter hinterlegt werden:

```
const BookModel = mongoose.model("Book", BookSchema, "BookCollection");
```

Der erste Parameter spielt dann keine Rolle mehr. Grundsätzlich gilt, wenn die Collection noch nicht auf der Datenbank existiert, wird sie automatisch erzeugt.

Es wird nun empfohlen, die Schemadefinition und die Generierung des Models in einem eigenen File auszulagern. Bspw. unter `Models/Book.js`:

```
const mongoose = require("mongoose");
const BookSchema = new mongoose.Schema( {
    title: {type: String},
    pages: {type: Number},
    authors: [{type: String}]
});
module.exports = mongoose.model("Book", BookSchema, "MyBooksCollection");
```

Die Nutzung erfolgt dann entsprechend, bspw. in `app.js`:

```
const BookModel = require("./Models/Book");
```

Um nun eine tatsächliche Instanz eines Buches zu speichern, müssen wir mit `create()` das Objekt und schließlich den Datensatz erstellen:

```
async function createNewBook() {
  try {
    // Erstellung des Objektes:
    const myBook = await BookModel.create({title:"C Language",
    pages:479,
    authors: ["Kernighan", "Ritchie"]});
    await myBook.save();  // hier wird das Objekt in der DB gespeichert
  } catch (error) {
    console.log("Fehlermeldung: " + error);
  }
}
createNewBook();
```

Da es sich bei `create()` und `save()` um Promises handelt, können wir diesen Vorgang in ein `async/await` Konstrukt einbinden. Eventuelle Fehlersituationen durch die Validierungen werden über den `try/catch` Block abgefangen. Die Validierung wird sowohl bei `create()`, als auch bei `save()` durchgeführt (Achtung, bei vielen anderen Manipulationen, wie bspw. `findAndUpdate()` nicht!) Nach jeder **späteren** Änderung im JavaScript Objekt müssen wir entsprechend `save()` aufrufen, damit die Daten auf der MongoDB landen.

Achtung: Es gibt Datentypen, bei denen vor `save()` die `Methode markModified()` aufgerufen werden muss. Hierzu zählen die Datentypen `Date` und `Mixed`. Haben wir bspw. im Schema ein Datum, bspw.:

```
released: {type: Date},
```

muss eine Änderung wie folgt ablaufen:

```
myBook.released.setMonth(10);
myBook.markModified("released");
await myBook.save();
```

Anmerkung: setMonth(10) würde den November setzen, da die Monatszählung bei 0 beginnt. Also Januar ist 0, Februar 1 usw.

4 Referenzen

Die Stärke von relationalen Datenbanken sind die Joins. MongoDB erlaubt dies zwar auch, jedoch nicht so performant. Trotzdem sind sie ein wichtiger Bestandteil jeglicher Datenbanken. Erzeugen wir ein eigenes Schema für unsere Autoren:

```
const mongoose = require("mongoose");
const AuthorSchema = new mongoose.Schema( {
    name: {type: String},
    birhtdate: {type: Date}
});
module.exports = mongoose.model("Author", AuthorSchema,
  "MyAuthorsCollection");
```

Und fügen die Autoren zu unserem Buchschema hinzu:

```
const mongoose = require("mongoose");
const BookSchema = new mongoose.Schema( {
    title: {type: String},
    pages: {type: Number},
    authors: [{type: mongoose.SchemaTypes.ObjectId, ref: "Author"}]
});
module.exports = mongoose.model("Book", BookSchema, "MyBooksCollection");
```

Als nächstes schreiben wir unsere beiden Autoren:

```
async function createNewAuthor(authorName, authorBirth) {
  try {
    const myBook = await AuthorModel.create(
      {name:authorName, birhtdate:authorBirth});
    await myBook.save();
  } catch (error) {
    console.log("Fehlermeldung: " + error);
  }
}
createNewAuthor("Kenrighan", "1942-01-01");
createNewAuthor("Ritchie","1941-09-09");
```

Und fügen die beiden Autoren in unser neues Buchobjekt ein:

```
const {ObjectID} = require("bson");
…
const myBook = await BookModel.create({title:"C Language",
pages:479,
authors: [ObjectID("634ee9925ce9a5c2a42e5a94"),
 ObjectID("634ee9925ce9a5c2a42e5a98")]});
await myBook.save();
…
```

5 Datenverarbeitung

Von den CRUD-Aktionen haben wir jetzt das „C“ wie Create besprochen. Nun müssen wir noch das Lesen, Ändern und Löschen ermöglichen. Hierbei ist die wichtigste Funktionalität, die Datensätze zu finden. Hierbei gibt es zwei wichtige Funktionen. Zuerst das Finden basierend der eindeutigen ID, welche in MongoDB immer existiert:

```
const book = await BookModel.findById("634d80f5563d39cf55c38ec7");
```

Da die ID eindeutig ist, liefert uns findById() exakt ein Objekt oder null. Daneben können wir auch mit der Funktion `find()` die gleichen Filterregeln anwenden wie in der MongoDB Shell:

```
const book = await BookModel.find({title: "C Language"});
```

Da `find()` nicht zwingend einen Eintrag findet, erhalten wir in der Variablen `book` ein Array der gefundenen Datensätze. Eine weitere Möglichkeit ist es, mit dem `where()` Syntax zu arbeiten. Hierbei können wir mehrere Bedingungen hintereinander platzieren:

```
const book = await
BookModel.where("title").equals("C Language").where("pages").gt(300);
```

Die Ergebnismenge wird von den nachgelagerten Funktionen entsprechend weiter gefiltert. Daneben können wir noch die Ausgabe limitieren und auch die auszugebenden Informationen einschränken (Stichwort „Projektion“):

```
const book = await
BookModel.where("title").equals("C Language").limit(2).select("pages");
```

Kümmern wir uns nun nochmal um die Referenzen. Wenn wir bspw. unser Buch nach der ID suchen und ausgeben, sehen wir anstatt den Autoren die ObjektIds, da wir sie ja im vorigen Kapitel als Referenz angegeben haben. Wollen wir die tatsächlichen Autoreninformationen sehen, so hilft uns die Funktion `populate()` weiter:

```
BookModel.findById("634eea973f50b32978909499").populate("authors");
```

Haben wir ein Objekt über eine der oben genannten Methoden gefunden, können wir es ändern und mit `save()` wieder abspeichern:

```
BookModel.findById("634eea973f50b32978909499");
book.title = "C-Language";
book.save();
```

Es gibt für das Update auch andere Methoden, jedoch greifen hier die Validierungsmethoden nicht, weshalb man üblicherweise den oben gezeigten Weg geht.

Schließlich haben wir noch die Möglichkeit, einen Datensatz zu löschen. Dies geschieht über `deleteOne()`:

```
await BookModel.findById("634d80f5563d39cf55c38ec7").deleteOne();
```

Daneben gibt es noch `deleteMany()`, wobei dies die Gefahr birgt, zu viele Daten zu löschen.