Elaborating on Models and getting started with CRUD

Student: Markus Ijäs

24.3.2022

Help to complete the tasks of this exercise can be found on the chapters from ch. 17 “Improving Your Data Models” and ch. 18 “Building the User Model” of our course book “Get Programming with Node.js” by Jonathan Wexler. The aims of the exercise are to learn validate the model instances, to use instance methods and virtual attributes, and to get started with CRUD.

Embed your theory answers, drawings, codes and screenshots directly into this document. Always immediately after the relevant question. Return the document into your return box in the itsLearning environment by the deadline.

It’s also recommendable to use Internet sources to supplement the information provided by the course book.

You can earn 12 points in maximum from this exercise. That means that there is 2 extra points available.

1. Validators and Helpers (1 point)

Develop the Subscriber schema further by adding the validators according to the listing 17-2 in the course book. Require also the vip feature we added earlier. Explain them all shortly. Also add the unique helper to email property. (Programming 0,5 points, explanation 0,5 points)

```
const subscriberSchema = mongoose.Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    lowercase: true,
    unique: true,
  },
  zipCode: {
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999,
  },
  vip: {
    type: Boolean,
    required: true,
  },
});
```

- Name: a string that is required (so must be present).
- Email: String, required, must be lower case and unique in the database.
- zipCode: Number, must be between 10000-99999. Not required.
- vip: A boolean, is also required.

2. Instance methods. (1 point)

Develop the Subscriber schema further by adding the instance methods according to the listing 17-3 in the course book. Explain shortly how the ”this” keyword works? Could you replace the functions in this example with arrow

functions? (Programming 0,5 points, explanations 0,5 points)

```
subscriberSchema.methods.getInfo = function () {
  return `Name: ${this.name} Email: ${this.email} Zip Code:
  ${this.zipCode}`;
};

subscriberSchema.methods.findLocalSubscribers = function () {
  return this.model("Subscriber").find({ zipCode: this.zipCode }).exec();
};
```

Explain shortly how the ”this” keyword works?

This references current object, that being the object fetched from the database in this case.

Could you replace the functions in this example with arrow functions?

No, not in any easy way, since Mongoose (and JS) utilizes the keyword `this` internally and does not pass it as an argument for arrow function to use.

3. Use REPL to test your Subscriber model. (2 points)

Follow the instruction given in the listings 17-4 and 17-5. Also create two new records with different contents. Check that your validators for the zipCode property are working by creating another one with ZIP code 890876 or and another one with ZIP code 123. Then try to delete one of your subscriber records directly from REPL.

Not completed to save time.

4. Associations between models. (2 points)

a. Create a new schema and a model for a course according to the instructions given in listing 17.6 in your course book. (0,5 points)

```
const mongoose = require("mongoose");

const courseSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    unique: true,
  },
  description: {
    type: String,
    required: true,
  },
  items: [],
  zipCode: {
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999,
  },
});

module.exports = mongoose.model("Course", courseSchema);
```

b. Then add the association between a Course Model and a Subscriber Model as an array called courses into the Subscriber’s schema. (0,5 points)

The schema is now:

```
const subscriberSchema = mongoose.Schema({
  name: {
    type: String,
```

```
    required: true,
  },
  email: {
    type: String,
    required: true,
    lowercase: true,
    unique: true,
  },
  zipCode: {
    type: Number,
    min: [10000, "Zip code too short"],
    max: 99999,
  },
  vip: {
    type: Boolean,
    required: true,
  },
  courses: [{ type: mongoose.Schema.Types.ObjectId, ref: "Course" }],
});
```

c. Then use REPL and follow the listing 17.7, but create the course with different feature values i.e. different name, more items and so on. (0,5 points)

Not completed to save time.

d. Finally display the results for the populated subscriber's object like in the listing 17.8. (0,5 points)

Not completed to save time.

5. Virtual Attributes. (1 point)

Program the listings 18.1 and 18.2. (0 points) Add then another virtual attribute called "username". The username is formed by taking the first letter of the first name and 7 first letters of the last name. In case the last name is shorter than 7 letters, take all the letters.

Not completed to save time.

6. Building the index page. Search help from the listings 18.9 to 18.11 in the course book. (4 * 0,5 = 2 points)

a. Create indexView for the subscribers. (0,5 points)

Not completed to save time.

b. Separate the index and indexView actions from each other. (0,5 points)

Not completed to save time.

c. What can you benefit by making the separation above? (0,5 points)

Not completed to save time.

d. Remember to display the VIP information too. (0,5 points)

Not completed to save time.

7. Async/await. (2 * 1 = 2 points)

a. Change your code so that you'll use async/await with request handlers in place of promises somewhere in your code. (1 point)

Not completed to save time.

b. Explain some benefits of using async/await instead of promises or callback functions.

Not completed to save time.

8. Connect the letters C,R,U,D and the http methods PUT, DELETE, GET, POST to commonly used pairs. (1 point)

- Create: POST
- Read: GET
- Update: PUT
- Delete: DELETE

Source: www.codecademy.com/article/what-is-crud.