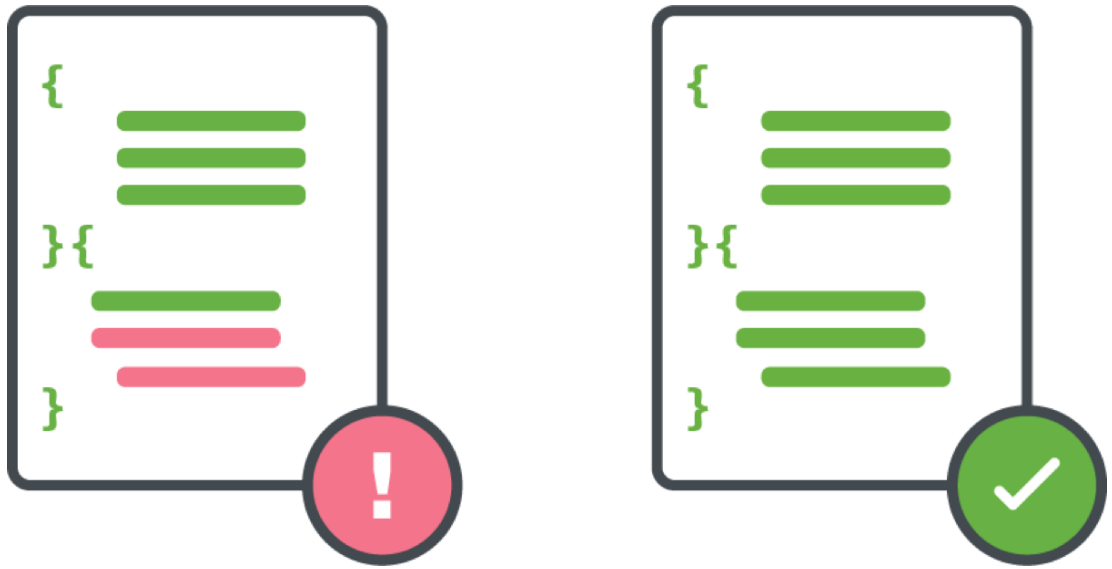
Node.js

Ch 8: Data Validation

hogeschool vives



D. HOSTENS & M. DIMA

— No validation on MongoDB side

- Last class we created a `course` Schema.
- All fields are optional!

```
const courseSchema = new mongoose.Schema({
  name: String,
  author: String,
  tags: [String],
  date: { type: Date, default: Date.now },
  isPublished: Boolean,
});
```

- This allows us to create a Course object

```
async function createCourse() {
  const course = new Course({
    name: "Angular Course",
    author: "M. Dima",
    tags: ["angular", "frontend"],
    isPublished: true,
  });
  const result = await course.save();
}
```

- Those course objects are not validated by MongoDB. We can as well create an empty object. Mongo does not validate!

```
async function createCourse() {
  const course = new Course({ });
  const result = await course.save();
}
```

— Mongoose Validation: required Fields as an example

- We can however build validation with Mongoose.
- Required

```
name: String,
```

- becomes

```
name: { type: String, required: true },
```

- If we try to create a course without a name

```
validator: [Function (anonymous)],
message: 'Path `name` is required.',
```

- Best practice: put all DB operation (bv. `save()`) inside a try-catch block

https://mongoosejs.com/docs/validation.html

— Mongoose `validate()` function

- Mongoose has a builtin validate() function

```
try{
    await course.validate();
    //const result = await course.save();
    //console.log(result);
  }
```

- Returns a Promise<void>. On error it jumps to the catch block
- Not very useful, it returns no result.
- A boolean as result would be a better outcome
- A workaround would be providing a callback function but this is not good practice

— Mongoose validation - not on DB level!

- This validation takes place in Mongoose. MongoDB does not care about this validation.
- In MS SQL or MySQL you can have validation on DB level. In MongoDB this does not exist!
- In previous classes we introduced Joi.
- This is not a replacement for Joi!
- We combine both methods
- Joi: first line validation on input/data from client
- After: Mongoose validation before we write to the DB

— Mongoose built-in validators

- We already handled one: Required
- *required* can be set to a *boolean* or be a function that returns a *boolean*. This is useful for conditional validation
- An example of a conditional *required*:
- Price is only *Required* if the course is *published*

```
const courseSchema = new mongoose.Schema({
  name: {type: String, required: true},
  ...
  isPublished: Boolean,
  price: {
    type: Number,
    required: function() { return this.isPublished; }
  }
});
```

- Please note!: we don't use an arrow function here! Arrow functions don't know *this*

— Mongoose built-in String validators
minlength, maxlength and match

- minlength= minimal length of a string
- maxlength= maximal length of a string
- match= a regex pattern (e.g. begins with A = `/^A/`)

```
const courseSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5,
    maxlength: 255,
    match: /pattern/    //replace pattern with regex
  },
  author: String,
  ...
});
```

— Mongoose built-in String validators: enum

- enum= a list of all possible values
- for example: several predefined categories

```
const courseSchema = new mongoose.Schema({
...
  category: {
    type: String,
    required: true,
    enum: ['web', 'mobile', 'network']
  },
...
});
```

— Mongoose custom validators

- Sometimes the builtin validators are not enough
- For instance our `tags` property
- Imagine: each course needs at least one tag

```
 tags: [String],
```

- wordt

```
const courseSchema = new mongoose.Schema({
  ...
  tags: {
    type: Array,
    validate: {
      validator: function (v){
        return v && v.length > 0;
      },
      message: 'A course should have at least one tag.'
    } // message is optional
  },
...
});
```

— Mongoose async validators

- What if your custom validator needs to read data from the DB?
- In that case we need an *Asynchronous validator*
- We adjust our previous custom validator

```
...
tags: {
    type: Array,
    validate: {
      isAsync: true,
      validator: function(v, callback){
        setTimeout(() => {
          const result = v && v.length > 0;
          callback(result);
        }, 1000);
      },
      message: 'A course should have at least one tag.'
    } // message is optional
  },
...
```

— Other interesting SchemaType properties

- With Strings
  - Lowercase (to Lower Case) `lowercase: true`
  - Uppercase (to Upper Case) `uppercase: true`
  - Trim (remove spaces around) `trim: true`
- Integer rounding (price)

```
price: {
    type: Number,
    required: function() { return this.isPublished; },
    min: 10,
    max: 200,
    get: v => Math.round(v),
    set: v => Math.round(v)
  }
```

- On creating a course the setter is called, on read, the getter

— Adding persistence to our API

- We continue with our Library application
- starter code on Github Classroom: see Toledo Ch8-Starter

- We replace the in-memory Genres kept in an Array by a Mongo DB persisted storage.

- We start with the Genres route

— Install Mongoose and connect

- inside the terminal: `npm install mongoose`
- please note the DB must run before the application:
- Connecting once inside `index.js`
- Require mongoose

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

- Connect with the DB

```
mongoose.connect('mongodb://localhost/vivesbib')
    .then(()=> console.log('Connected to
MongoDB'))
    .catch(err => console.log('Could not connect
to MongoDB...'));
```

— Editing the Genres Route

- Import Mongoose

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

- Define Schema

```
const genreSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5,
    maxlength: 50
  }
});
```

- From that schema we create a model

```
const Genre = mongoose.model('Genre', genreSchema);
```

— Schema and model refactoring

- Given genreSchema is only used once we can merge the schema and model in one code block:

```
const Genre = mongoose.model('Genre', new
mongoose.Schema({
  name: {
    type: String,
    required: true,
    minlength: 5,
    maxlength: 50
  }
}));
```

- After that we can delete our genres Array

— Editing the Endpoints: get /

```
router.get('/', (req, res) => {
  res.send(genres);
});
```

- becomes

```
router.get('/', async (req, res) => {
  const genres = await
Genre.find().sort('name');
  res.send(genres);
});
```

Editing the Endpoints: post /

```
router.post('/', (req, res) => {
  const { error } = validateGenre(req.body);
  if (error) return
res.status(400).send(error.details[0].message);
  const genre = {
    id: genres.length + 1,
    name: req.body.name
  };
  genres.push(genre);
  res.send(genre);
});
```

- becomes

```
router.post('/', async (req, res) => {
  const { error } = validateGenre(req.body);
  if (error) return
res.status(400).send(error.details[0].message);
  let genre = new Genre({name: req.body.name});
  genre = await genre.save();
  res.send(genre);
});
```

Editing the Endpoints: put /:id

```
router.put('/:id', (req, res) => {
  const genre = genres.find(c => c.id === 
parseInt(req.params.id));
  if (!genre) return res.status(404).send('The genre with the 
given ID was not found.');
  const { error } = validateGenre(req.body); 
  if (error) return 
res.status(400).send(error.details[0].message);
    genre.name = req.body.name; 
  res.send(genre);
});
```

- becomes (we use the *update first* approach)

```
router.put('/:id', async (req, res) => {
  const { error } = validateGenre(req.body); 
  if (error) return 
res.status(400).send(error.details[0].message);
  const genre = await Genre.findByIdAndUpdate(req.params.id, 
{name: req.body.name},{new: true});
  if (!genre) return res.status(404).send('The genre with the 
given ID was not found.');
  res.send(genre);
});
```

Editing the Endpoints: delete /:id

```
router.delete('/:id', (req, res) => {
  const genre = genres.find(c => c.id === 
parseInt(req.params.id));
  if (!genre) return res.status(404).send('The genre with the 
given ID was not found.');
  const index = genres.indexOf(genre);
  genres.splice(index, 1);
  res.send(genre);
});
```

- becomes

```
router.delete('/:id', async (req, res) => {
  const genre = await Genre.findByIdAndDelete(req.params.id);
//vroeger findByIdandRemove()
  if (!genre) return res.status(404).send('The genre with the 
given ID was not found.');
  res.send(genre);
});
```

— Editing the Endpoints: get /:id

```
router.get('/:id', (req, res) => {
  const genre = genres.find(c => c.id === parseInt(req.params.id));
  if (!genre) return res.status(404).send('The genre with the given ID was not found.');
  res.send(genre);
});
```

- wordt

```
router.get('/:id', async (req, res) => {
  const genre = await Genre.findById(req.params.id);
  if (!genre) return res.status(404).send('The genre with the given ID was not found.');
  res.send(genre);
});
```

— Testing the Application

- We run the applicatie (by default `index.js` is launched):

```
nodemon .
```

- Inside the browser we check the nex URL: http://localhost:3000/api/genres
- We get an empty array [] DB is empty
- In Postman we create a genre:
- POST request, URL: http://localhost:3000/api/genres/, Body > RAW > JSON

```
{
   "name": "sci-fi"
}
```

- We refresh the browser:

```
[{"_id":"60660697a67cea840cebd666","name":"sci-fi","__v":0}]
Connected to MongoDB
```

API call in Rest Client: api-calls/genres.http


```
@hostname = http://localhost
@port = 3000
@host = {{hostname}}:{{port}}
@contentType = application/json
@url = {{host}}/api/genres


GET {{url}}

###

POST {{url}}
Content-Type: {{contentType}}

{
    "name": "new course"
}
```

— Lab

- Assignment link: see Toledo