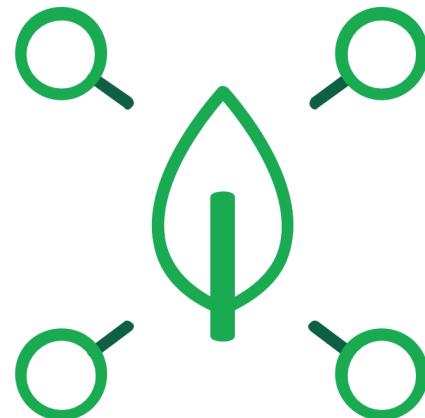


Node.js

Les 9 - Mongoose Modeling

M. Dima



```
{  
  name : 'left-handed smoke shifter',  
  manufacturer : 'Acme Corp',  
  catalog_number: 1234,  
  parts : [  
    {  
      _id : ObjectId('AAAA'),  
      partno : '123-aff-456',  
      name : '#4 grommet',  
      qty: 94,  
      cost: 0.94,  
      price: 3.99  
    },  
    {  
      _id : ObjectId('BBB'),  
      partno : '425-EFF-123',  
      name : '#8 Frombet',  
      qty: 13,  
      cost: 0.34,  
      price: 7.99  
    }  
  ]  
}
```

— How to model data?

“ What is used together in the application
is stored together in the database.

— Data modeling

- Bij NoSQL databanken hebben we drie opties:
- **Optie 1:** Referencing (**normalization**) Bv. in een "boek"-document uit de collectie "boeken" verwijzen naar een "auteur"-document uit de collectie "auteurs"
- **Optie 2:** Embedding (**denormalisation**) BV. Informatie over de auteur wordt opgenomen in het boek document in de collectie boeken.
- **Optie 3:** Hybride. Het boek bevat een referentie naar een auteur document maar ook een property naam die dezelfde kan zijn als dat van het document `auteur` of kan voor dit boek document specifiek een andere waarde bevatten.
- Afhankelijk van onze relaties gebruiken we **Referencing**, **Embedding** of **Hybride**

— Data Relaties

- **1 : 1** => Elke auteur heeft één en slechts een boek geschreven
- **Many : Many** => Elk boek heeft verschillende auteurs en elk auteur heeft verschillende boeken geschreven
- **1 : Many** => Elke auteur heeft verschillende boeken geschreven
- Dit kunnen we verder onderverdelen
- **1 : Few** => Elke auteur heeft enkele boeken geschreven
- **1 : Many** => Elk boek heeft veel reviews
- **1 : Ton** => Elke chat heeft miljoenen berichten

— Wat gebruiken we en wanneer?

- Opgelat bij NoSQL geen data-integriteit door relaties
- Referenties worden niet automatisch aangepast
- Bij **Referencing** heb je **meer queries** nodig als je bij de boek-data ook de auteur-data wilt ophalen
- Bij **Embedding** heb je beide stukken data met **één query** maar wel meer dubbele data en bij aanpassingen moet je dat overal aanpassen
- **1 : Few => embedding**
- **1 : Many =>** Bij vooral data uitlezen *embedding* en bij data die vaak wijzigt *referencing* (we queryen slechts een deel van de data)
- **1 : Ton & Many : Many => referencing**

— Wat gebruiken we en wanneer?

Reference

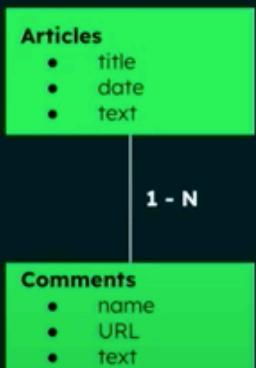
1. when the "many" side is a huge number
2. for integrity on write operations on many-to-many
3. when a piece is frequently used, but not the other and memory is an issue

Embed

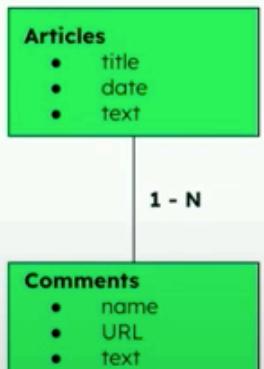
1. for integrity of read operations
2. for integrity of write operations on one-to-one and one-to-many
3. for data that is deleted or archived together
4. by default

— Referencing & Embedding

Tabular/Relational



Reference/Link is two collections in MongoDB



Embedding is using one collection in MongoDB



— Referencing & Embedding

- Trade Off tussen consistentie en performantie

```
//Referencing (Normalization)
//HIGH Consistency LOW Performance
let author = {
    name: 'Vives'
}
let course = {
    author: 'id'
}
```

```
//Embedding documents (Denormalization)
//HIGH Performance LOW Consistency
let course = {
    author: {
        name: 'Vives'
    }
}
```

— Hybride aanpak

- Bv elk "*author*"-document heeft 50 *properties*
- Course bevat een author document maar met een referentie en één of enkele van de properties
- Ook handig voor een **snapshot in tijd** (bv prijs van product op het moment dat het besteld werd)

```
//Hybrid
let author = {
    name: 'Vives'
    //50 other properties
}
let course = {
    author: {
        id: 'ref',
        name: 'Vives'
    }
}
```

— Referencing documents

- starterfiles: [GitHub](#) => `referencing.js` & `embedding.js`

```
const Author = mongoose.model('Author', new mongoose.Schema({  
    name: String,  
    bio: String,  
    website: String  
}));  
const Course = mongoose.model('Course', new mongoose.Schema({  
    name: String,  
}));
```

- De functie `createCourse` neemt twee parameters: een naam en een author-id

```
async function createCourse(name, author) {  
    const course = new Course({  
        name,  
        author  
   });
```

— Schema als blauwdruk

- schema = document data structure
- model = interface tussen app en db
- we maken een auteur en een course aan

```
createAuthor('Vives', 'My bio', 'My Website');
..._id: 60772ac8684c78c4435f4153,
  name: 'Vives', ...
createCourse('Node Course', '60772ac8684c78c4435f4153')
{ _id: 60772af0f3915ac45d4064ce, name: 'Node Course', __v: 0 }
```

- er werd geen referentie naar de author opgeslagen!
- schema bevat geen author referentie!

```
const Course = mongoose.model('Course', new mongoose.Schema({
  name: String
}));
```

— Schema aanpassen

```
const Course = mongoose.model('Course', new
mongoose.Schema({
  name: String,
  author: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Author' //target collection
  }
}));
```

- We maken opnieuw een document aan

```
createCourse('Node Course', '60772ac8684c78c4435f4153')
{
  _id: 60772d2675c702c574002c58,
  name: 'Node Course',
  author: 60772ac8684c78c4435f4153,
  __v: 0
}
```

- Nu bevat het document de referentie naar de author

— Courses ophalen

- We halen alle courses op

```
async function listCourses() {  
  const courses = await Course  
    .find()  
    .select('name');  
  console.log(courses);  
}  
...  
listCourses();  
{  
  _id: 60772f4ac904a3c6491bbf46,  
  name: 'Node Course',  
  author: 60772ac8684c78c4435f4153,  
  __v: 0  
}
```

- Onze cursussen bevatten enkel een id van de author maar niet de naam en andere properties
- Wat als we de andere gegevens ook willen?

— populate() functie

- Omdat er in ons course-schema een referentie gemaakt werd naar de collectie Authors, kunnen we vragen aan mongoose om achter de schermen de gegevens van de author op te halen met de functie populate

```
async function listCourses() {  
    const courses = await Course  
        .find()  
        .populate('author') ←  
        .select('name');  
    console.log(courses);  
}  
  
{  
    _id: 60772f4ac904a3c6491bbf46,  
    name: 'Node Course',  
    author: [  
        _id: 60772ac8684c78c4435f4153,  
        name: 'Vives',  
        bio: 'My bio',  
        website: 'My Website',  
        ...  
    ]  
}
```

— populate() argumenten

- de functie populate neemt als tweede argument de properties die het moet tonen

```
  .populate('author', 'name')
name: 'Node Course',
  author: { _id: 60772ac8684c78c4435f4153,
name: 'Vives' }
```

- name zonder de _id

```
  .populate('author', 'name _id')
name: 'Node Course',
  author: { name: 'Vives' }
```

- name zonder de _id wel met bio

```
  .populate('author', 'name _id bio')
```

- populate functies kan je *chainen* om verschillende collecties na elkaar te queryen

— Embedding documents

- starterfiles: [GitHub](#) => referencing.js & embedding.js

```
const authorSchema = new mongoose.Schema({  
    name: String,  
    bio: String,  
    website: String  
});  
const Author = mongoose.model('Author', authorSchema);  
const Course = mongoose.model('Course', new mongoose.Schema({  
    name: String  
}));
```

- we nemen in ons course-object een author object op

```
const Course = mongoose.model('Course', new mongoose.Schema({  
    name: String,  
    author: authorSchema  
}));
```



geen ObjectID als bij ref.

- Author binnen course is een (volwaardig) subdocument. De meeste document features zijn ok hiervoor valid: bv validatie

— Course aanmaken

- We deleten de db playground (schone lei)
- We maken een course object aan

```
async function createCourse(name, author) {  
  const course = new Course({  
    name,  
    author  
});  
...  
createCourse('Node Course', new Author({ name:  
  'Vives' }));  
$ node embedding.js  
{  
  _id: 6077363c35b1d0c8b49ef261,  
  name: 'Node Course',  
  author: { _id: 6077363c35b1d0c8b49ef260, name:  
  'Vives' },  
  __v: 0  
}
```

— Author aanpassen

- het author-subdocument kan niet op zijn eigen opgeslagen worden enkel in de context van zijn parent
- we schrijven een nieuwe functie updateAuthor()

```
async function updateAuthor(courseId){  
    const course = await Course.findById(courseId);  
    course.author.name = 'M. Dima';  
    course.save(); //course.author.save() bestaat niet!  
}  
updateAuthor('60798f05a3a949f04af32ab6');
```

In MONGODB Compass

```
{"_id": {"$oid": "60798f05a3a949f04af32ab6"},  
"name": "Node Course",  
"author": {  
    "_id": {"$oid": "60798f05a3a949f04af32ab5"},  
    "name": "M. Dima"},  
    "v": 0  
}
```

— Author aanpassen volgens de 'Update First' methode

- We kunnen ook onze author updaten zonder eerst te Queryen en dan het document opnieuw op te slaan:

```
async function updateAuthor(courseId){  
    const course = await Course.findByIdAndUpdate  
({_id: courseId}, {  
        $set: {  
            'author.name': 'M. Dima'  
        }  
    } );  
}  
updateAuthor('60798f05a3a949f04af32ab6');
```

- property of subdocument verwijderen met \$unset

```
$unset: {  
    'author': ''  
}
```

— Validatie: required

- Huidige toestand author subdocument

```
const Course = mongoose.model('Course', new
mongoose.Schema({
  name: String,
  author: authorSchema
}));
```

- We passen die aan naar *required*

```
const Course = mongoose.model('Course', new
mongoose.Schema({
  name: String,
  author: {
    type: authorSchema,
    required: true
  }
}));
```

- Indien andere author property required is dan aanpassen in authorSchema zelf

— Array van subdocumenten

- Wat als een course verschillende authors heeft?

```
const Course = mongoose.model(  
  "Course", [ ] ,  
  new mongoose.Schema({  
    name: String, [ ]  
    authors: [authorSchema], [ ]  
  }) [ ]  
) ; [ ]  
  
async function createCourse(name, authors) {  
  const course = new Course({  
    name, [ ]  
    authors, [ ]  
  }) ; [ ]  
  ... [ ]  
  createCourse('Node Course', [ ]  
    new Author({ name: 'Vives' }), [ ]  
    new Author({ name: 'M. Dima' })) [ ]  
} ); [ ]
```

— Authors toevoegen en verwijderen uit array

- we maken een nieuwe functie aan `addAuthor()`

```
async function addAuthor(courseId, author){  
    const course = await Course.findById(courseId);  
    course.authors.push(author);  
    course.save();  
}  
  
addAuthor('6079a25f42697ff398cb9de0', new Author({ name: 'Milan D.' }));
```

- we maken een nieuwe functie aan `removeAuthor()`

```
async function removeAuthor(courseId, authorId){  
    const course = await Course.findById(courseId);  
    const index = course.authors.findIndex(obj => obj._id ==  
authorId);  
    course.authors.splice(index, 1);  
    course.save(); //Opgelat vorige versies author.save()  
}  
  
removeAuthor('6079a25f42697ff398cb9de0', '6079a4a166aa79f40bee78d0')
```

— Object ID's in MongoDB

- Elk in mongo opgeslagen document krijgt een unieke id
- Formaat 24 symbolen (hex) = 12 bytes
- voorbeeld `6079a4a166aa79f40bee78d0`
- Eerste 4 bytes bevatten de timestamp: `6079a4a1`
- Documenten op ID sorteren = op creatiedatum sorteren
- Volgende 3 bytes = machine identifier `66aa79`
- Volgende twee bytes = process identifier `f40b`
- Laatste 3 bytes = counter (max $2^{24} = 16M$) `ee78d0`
- Id wordt gegenereerd door de driver en niet door MongoDB! d.w.z we hoeven niet te wachten op de db
- => schaalbare applicaties: verschillende instanties moeten niet overleggen met de DB voor een unieke ID

— Zelf een ID genereren

```
const mongoose = require('mongoose');
const id = new mongoose.Types.ObjectId();
console.log(id);
milan@les9 ~ % node objectids.js
6079b1b772f86ef6a42b7c37
```

- Timestamp er uithalen met `getTimestamp()`

```
console.log(id.getTimestamp());
milan@les9 ~ % node objectids.js
2021-04-16T15:50:21.000Z
```

- ID valideren met `mongoose.Types.ObjectId.isValid()`

```
const isValid =
mongoose.Types.ObjectId.isValid(id);
console.log(isValid);
milan@les9 ~ % node objectids.js
true
```

— Transactions

- In MongoDB geen automatische roll-back
- Als er bv tussen twee DB transacties iets verkeerd loopt
- We gebruiken daarvoor een package Fawn

```
const Fawn = require('fawn');
...
try {
  new Fawn.Task()
    .save('rentals', rental)
    .update('books', { _id: book._id }, {
      $inc: { numberInStock: -1 }
    })
    .run();
  res.send(rental);
}
catch(ex) {
  res.status(500).send('Something failed.');
}
```

Transactions

- <https://www.npmjs.com/package/fawn>

```
fawn-transactions.js

var mongoose = require("mongoose");
mongoose.connect("mongodb://127.0.0.1:27017/testDB");
Fawn.init(mongoose);

var task = Fawn.Task();
//assuming "Accounts" is the Accounts collection
task.update("Accounts", {firstName: "John", lastName: "Smith"}, {$inc: {balance: -20}})
  .update("Accounts", {firstName: "Broke", lastName: "Ass"}, {$inc: {balance: 20}})
  .run()
  .then(function(results){
    // task is complete

    // result from first operation
    var firstUpdateResult = results[0];

    // result from second operation
    var secondUpdateResult = results[1];
  })
  .catch(function(err){
    // Everything has been rolled back.

    // log the error which caused the failure
    console.log(err);
  });
}
```

— Code les & Opdracht

- Oplossingen code les: [referencing](#), [embedding](#), [obj_id](#)

Opdracht

- Vervolledig de code in de applicatie (zie comments)
- assignment link: zie Toledo