MongoDB

javascript et son écosystème

Licence mention Informatique
Université Lille – Sciences et Technologies

MongoDB

- une système de BD orienté **documents**
  terminologie *mongodb* :
  - entrée dans la base = **document**
  - ensemble de documents = **collection** (≡ table)
- les données/documents sont stockés au format BSON
- à chaque donnée est associée un identifiant unique `_id`
- MongoDB ne gère pas les transactions

serveur

- lancement du serveur

  `mongod --dbpath` *chemin pour les bases*
- lancement port 27017 par défaut

mongo

- shell client : `mongo`
- `show dbs` pour lister les bases existantes
- `use baseName` pour utiliser (et créer) une base
- `show collections` pour lister les collections de la base

CRUD operations

- db.*collectionName*.find(*args*)
- db.*collectionName*.insert(*args*)
- db.*collectionName*.update(*args*)
- db.*collectionName*.delete(*args*)

exemples

mongodb v0

```
use dbbooks
show collections
db.books.find()
db.books.insert({title : 'The Lord of the Ring', author : 'JRRT', year : 1954})
db.books.find()
db.books.insert({ title : 'Dune',  author : 'Herbert Frank', year : 1965 })
db.books.find().pretty()
db.books.find({year : 1954})
db.books.insert({ title : 'Bilbo',  author : 'Jrrt', year : 1954 })
db.books.update({ title : 'Bilbo'}, { $set : {author : 'JRRT', year : 1937} })
db.books.find({author : 'JRRT'})
db.books.update({ title : 'Bilbo'}, { $set : { type : 'novel'} })
db.books.find({author : 'JRRT'})
db.books.deleteOne({author : 'Herbert Frank'})
db.books.deleteMany({author : 'JRRT'})
```

import de données :

```
mongoimport --db dbbooks --collection books --type csv
                          --file ./books.csv --headerline
```

```
db.books.find({year : { $gt : 1980 }}).pretty()
```

mongoose

- par les créateurs de MongoDb

  elegant MongoDb object modeling for Node.js
- permet de définir des modèles (schémas) pour les documents
- facilite la gestion des connexions
- Mongoose fait l'interface entre Node.js et MongoDb

  Mongoose = Object Data Modeler for Node.js
- installation : `npm install mongoose --save`

mongodb v0.5

mise en œuvre

- accéder à Mongoose :

  mongodb v1

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

- établir une connexion :
  - URI de connexion :
    `mongodb://username:password@host:port/database`
    - `username`, `password` et `port` sont optionnels
  - création de la connexion, deux possibilités :
    - `mongoose.connect(`*URI*`, {useNewUrlParser: true, , useUnifiedTopology : true})`
      connexion définie dans `mongoose.connection`
    - `const dbConn = mongoose.createConnection(`*URI*`, `*options*`)`
      permet de créer plusieurs connexions et d'exporter `dbConn`
  - gérer les événements de connexion et la déconnexion
    - `dbConnection.on('connected'/'error'/'disconnect',`*callback*`)`
      `mongoose.connect.on('connected',` *callback*`)` si premier cas
    - `dbConnection.close(` *callback* `)`
      gérer les différentes situations de fin de processus

bonne pratique : définir un contrôleur pour gérer la BD

```
// dans /controllers/db.js
const mongoose = require('mongoose');

const dbURI = 'mongodb://localhost/dbbooks';
const dbConnection
          = mongoose.createConnection(dbURI, { useNewUrlParser: true });

module.exports = dbConnection;      // export de la connexion créée

dbConnection.on('connected', ... );
dbConnection.on('disconnected', ... );
dbConnection.on('error', ... );

// process ends
process.on('SIGINT', ... );         // application killed (ctrl+c)
process.once('SIGUSR2', ...);       // pour nodemon
process.on('SIGTERM', ...);         // process killed (POSIX)
```

définition du modèle de données

- définir comment les données sont structurées
  - définition d'un document = **schéma**
  - une entrée dans un schéma = **path**
  - un *schéma* est constitué d'un ensemble de *path*

  bonne pratique : définir un dossier `models` pour y placer les définitions des schémas

- un schéma doit être « compilé » en un **modèle** pour son utilisation dans l'application

schéma

- déclaration d'un schéma

mongodb v2

```
const theSchema = new mongoose.Schema( { les path du schéma }
                                       );
```

```
// dans /models/books.js
const mongoose = require('mongoose');
const bookSchema = new mongoose.Schema( {
  title : String,
  author : String,
  cover : String,
  year : Number
} );
```

pas de *path* pour `_id`

- path :

pathName : objet_propriétés

```
title : { type : String }
```

écriture simplifiée quand il n'y a que la propriété `type`

path

- 8 types possibles
  `String`, `Number`, `Date`, `Boolean`, `Buffer`, `Mixed`, `Array`, `ObjectId`
  le type tableau se déclare en plaçant le type des élément entre crochets
- d'autres schémas peuvent être utilisés pour définir le type des sous-documents
- différentes options : `default`, `required`, `set`, `get`, ...
  + des options spécifiques aux différents types
    - `Number` : `min`, `max`
    - `String` : `lowercase`, ...

```
// dans /models/address.js
const addressSchema = new mongoose.Schema({
    number : { type : Number, min : 1 },
    street : String,
    zip : Number,
    town : { type : String, required : true}
});
module.exports = addressSchema;
```

```
// dans models/person.js
const addressSchema = require('./address');

const personSchema = new mongoose.Schema({
    name : { type : String, required : true },
    surnames : [String],
    age : { type : Number, default : 18, set : v => Math.floor(v) },
    birth : Date,
    address : addressSchema
});
module.exports = personSchema;
```

modèles

- pour être utilisé un schéma doit être « *compilé* » en un **modèle**
- le modèle permet les interactions avec une collection
- on utilise la connection pour créer le modèle et le lier à une collection

  theConnection.model(modelName, schema, collectionName)

```
// dans models/books.js
const dbConnection = require('../controllers/db');

const bookSchema = new mongoose.Schema({...});

const Books = dbConnection.model('Book',bookSchema,'books');

module.exports.schema = bookSchema;
module.exports.model = Books;
```

interactions avec les collections

- on utilise le modèle pour manipuler la collection à laquelle il est attaché
- les opérations sont des **promesses**
- opérations CRUD de base :
  - `create`
  - `find`
  - `update`
  - `remove`

find()

- récupérer des documents dans une collection

mongodb v3

```
// dans /controllers/books.js
const Books = require('../models/books').model; // on récupère le modèle
var list =
  (req, res) =>
      Books.find()
           .then( allBooks => res.render('books',
                                         { title : 'Book list',
                                           books : allBooks } ) );
```

mongodb v3.1

```
  Books.findOne()                          // /books/one
  Books.find({title : 'Dune'})             // /books/dune
  Books.find({year : {$gt : 2000} })       // /books/yearv1
  Books.find().where('year').gt(2000)      // /books/yearv2
```

findById

mongodb v3.2

```
// dans /controllers/books.js
  Books.findOne( { _id : req.params.bookId } )    // /books/detailsv1

  Books.findById( req.params.bookId )             // /books/detailsv2
```

create()

- création d'une nouvelle entrée dans une collection

```
let newBook = { title : value, author : value,
                year : value,  cover : value  };
Books.create(newBook)
     .then( newBook => ... );
```

mise en œuvre

mongodb v4

- serveur : deux routes : pour le « formulaire » et la requête de création

```
// routes/books.js
router.get('/create', booksController.createHome );
router.post('/create', booksController.create );
```

API REST :

méthode	utilisation	Response
POST	**C**reate new data	nouvel objet créé
GET	**R**ead data	objet réponse de la requête
PUT	**U**pdate document	objet mis à jour
DELETE	**D**elete document	`null`

- mise en place de la vue `/views/createBook.pug`
  qui charge côté client le script `/javascripts/create-book.js`

côté client

- côté client : mise en place de la requête

```
// dans /public/javascripts/create-book.js
const createBook =
  () => {
    let newBook = { title : title.value , author : author.value,
                    year : pubyear.value,  cover : cover.value };
    let body = JSON.stringify(newBook);
    let requestOptions = { method :  'POST',
      headers:{"Content-Type":"application/json"},
                body :  body };
      fetch('http://127.0.0.1:3000/books/create', requestOptions)
         .then( response => response.json() )
         .then( book => result.textContent = `book with id
                                              ${book._id} created`);
  }
```

- côté serveur : mise en place des contrôleurs

```
// dans /controllers/books.js
/* controller for GET /create */
const createHome =
  (req,res) => res.render('createBook');      // not REST

/* controller for POST /create */
const createBook =
  (req, res, err) => {
    //let newBook = { title : req.body.title, author : req.body.author,
    //                year : req.body.year, cover : req.body.cover };
    let newBook =  { ...req.body } ;
    Books.create(newBook)
      .then( newBook => res.status(200).json(newBook) );
  }
```

validation du schéma

mongodb v4.1

- validation des données par le schéma avant tout ajout de données

```
// dans /models/books.js
var setDefaultCover = cover => { ... }
var bookSchema = new mongoose.Schema({
  title : { type : String, required :  true },
  author : String,
  cover : { type : String, set :  setDefaultCover },
  year : Number
});
```

- gestion des erreurs dans la requête

```
// dans /controllers/books.js
Books.create(newBook)
  .then( newBook => res.status(200).json(newBook) )
  .catch( error => res.status(400).json(error) );
```

traitement de la requête

côté client... prise en compte de l'erreur

```
// dans /public/javascripts/create-book.js
fetch('http://127.0.0.1:3000/books/create', requestOptions)
  .then ( response => {
      if (response.ok)
        {  return response.json();  }
      else {  throw new Error(` creation failed ` ); }
    })
  .then(book => result.textContent = `book with id ${book._id} created` )
  .catch( error => result.textContent = `error :  ${error.message}` );
```

gestion du message d'erreur

mongodb v4.2

- exploitation de l'erreur fournie dans `response`

```
// dans /public/javascripts/create-book.js
fetch('http://127.0.0.1:3000/books/create', requestOptions)
  .then(response =>
            response.json()
               .then(json => ({ ok :  response.ok, json :  json}) )
  .then( response => {
      if (response.ok)
        { return response.json; }
      else
        { throw new Error(`creation failed : ${response.json.message}`);}
    })
  .then(book => result.textContent = `book with id ${book._id} created`))
  .catch( error => result.textContent = `error : ${error.message}` );
```

save() / création de sous-document

mongodb v4.3

- il faut créer le sous-document et sauvegarder le document parent
- le résultat de save() est la totalité du document parent

```
// dans /routes/books.js
router.post('/adddetails/:bookId', booksController.addDetails);

// dans /controllers/books.js
const addDetails =
  (req, res, err) => {
    let details = { ...req.body };  // les données sont dans req.body
    Books.findById( req.params.bookId )
      .then( book => {
                      book.details = details;
                      return book.save();
                    })
      .then( book => res.status(200).json(book.details))
      .catch( error => res.status(400).json(error) );
  }
```

update()

mongodb v5

- mise à jour d'un document
- méthode PUT

```
// dans /routes/books.js
router.get('/update/:bookId', booksController.updateForm );
router.put('/update/:bookId', booksController.update );
```

- côté client

```
// dans /public/javascripts/update-book.js
const updateBook =
  bookId => {
    let newBook = { title : title.value, author : author.value, ... };
    let body = JSON.stringify(newBook);
    let requestOptions = { method :'PUT', body : body,
                           headers: {"Content-Type""application/json"}};
    fetch(`http://127.0.0.1:3000/books/update/${bookId}`, requestOptions)
      .then ( response => ...
```

- côté serveur

```
// dans /controllers/books.js
const update =
  (req,res,err) => {
    let updatedBook = { ...req.body };
//  Books.findById( req.params.bookId ).update(updatedBook)
    Books.findByIdAndUpdate( req.params.bookId, updatedBook )
      .then( () => res.status(200).json({id : req.params.bookId}) )
      .catch( error => res.status(400).json(error) );
  }
```

- `findByIdAndUpdate(theId, thevalues, { new : true } )` permet d'avoir en résultat le document après modification
- `updateOne()`, `updateMany()`

remove()

mongodb v6

- suppression d'un document de la collection

```
// dans /routes/books.js
router.get('/delete/:bookId', booksController.delete );

// dans /controllers/books.js
const deleteBook =
  (req,res,err) => {
    //Books.findById( req.params.bookId ).remove()
    Books.findByIdAndRemove( req.params.bookId )
      .then( () => res.redirect('/books') )
      .catch( error => { throw error } );
  }
```

- REST : utiliser la méthode `DELETE`
- `deleteOne()`, `deleteMany()`

API REST

méthode	utilisation	Response
POST	**C**reate new data	nouvel objet créé
GET	**R**ead data	objet réponse de la requête
PUT	**U**pdate document	objet mis à jour
DELETE	**D**elete document	`null`

```
// dans /routes/bookrest.js
router.get('/', controller.home );
router.get( '/:bookId', controller.getBook );
router.post( '/', controller.createBook );
router.put( '/:bookId', controller.updateBook );
router.delete( '/:bookId', controller.deleteBook );
```

mongodb v7

```
// dans /controllers/bookrest.js
const getBook =
  (req,res) =>
    Books.findById( req.params.bookId )
         .then( book => res.status(200).json(book) );
const createBook =
  (req,res) => {
    let newBook = { ...req.body };
    Books.create(newBook)
         .then( book => res.status(200).json(book) );
  }
const updateBook =
  (req, res) => {
    let updatedBook = { ...req.body };
    Books.findByIdAndUpdate( req.params.bookId, updatedBook, { new : true } )
         .then( book => res.status(200).json(book) );
  }
const deleteBook =
  (req,res) =>
      Books.findByIdAndRemove( req.params.bookId )
           .then( () => res.status(200).end() );
```

côté client

```
// dans /public/javascripts/usebookrest.js
const getBook = bookId => {
      fetch(‘http://127.0.0.1:3000/bookrest/${bookId}‘, { method :’GET’ })
        .then( response => response.json() )
        .then( book => ... ) ...    }
const createBook = () => {
      let newBook = ...;  let body = JSON.stringify(newBook);
      let requestOptions = { method :’POST’,  body : body , headers : ... };
      fetch(‘http://127.0.0.1:3000/bookrest/‘, requestOptions)
        .then( response => response.json() )
        .then( book => ... ) ...    }
const updateBook = bookId => {
      let book = ...; let body = JSON.stringify(book);
      let requestOptions = { method :’PUT’, body : body , headers : ... };
      fetch(‘http://127.0.0.1:3000/bookrest/${bookId}‘, requestOptions)
        .then( response => response.json() )
        .then( book => ... ) ... }
const deleteBook = bookId => {
      fetch(‘http://127.0.0.1:3000/bookrest/${bookId}‘, { method :’DELETE’ })
        .then( () => ... ) ...    }
```