Лабораторна робота № 6

Тема: Основні типи та функції доступу до БД **MongoDB**

Мета: Засвоїти елементи створення, модифікації, читання та занесення даних з таблиць БД засобами Node.js

База даних MongoDB

Зараз з'являється все більше high-load проектів, що оперують великим обсягом даних. І вже не можна обійтись класичною реляційною моделлю зберігання цієї інформації. Все більш популярними стають бази даних NoSQL (NoSQL - позначає Not only SQL). Однією з таких баз даних є MongoDB

`MongoDB` – це база даних, яка зберігає ваші дані у вигляді документів. Як правило, ці документи мають `JSON` (* JavaScript Object Notation - текстовий формат обміну даними, заснований на `JavaScript`. Тут і далі прям. пер.) - подібну структуру:

```
{
  firstName: "Jamie",
  lastName: "Munro"
}
```

Далі документ має бути всередині колекції. Наприклад, у наведеному вище прикладі документа визначається об'єкт user. Далі цей об'єкт user став, швидше за все, частиною колекції під назвою users.

Одна з основних особливостей `MongoDB` – гнучкість структури її даних. Незважаючи на те, що в першому прикладі об'єкт user мав властивості firstName і lastName, ці властивості можуть бути відсутні в інших документах user колекції users. Саме це відрізняє `MongoDB` від баз даних `SQL` (* structured query language - мова структурованих запитів), наприклад, `MySQL` або `Microsoft SQL Server`, в яких для кожного об'єкта, що зберігається в базі даних, потрібна фіксована схема.

За рахунок здатності створювати динамічні об'єкти, які зберігаються у вигляді документів у базі даних, опрацьовує `Mongoose`

Що таке Mongoose?

`Mongoose` – це ODM (* Object Document Mapper – об'єктно-документний відтворювач). Це означає, що `Mongoose` дозволяє визначати об’єкти зі строго-типізованою схемою, що відповідає документу `MongoDB`.

`Mongoose` надає величезний набір функціональних можливостей для створення та роботи зі схемами. На даний момент `Mongoose` містить вісім `SchemaTypes` (* типи даних схеми), які можуть мати властивість, що зберігається в `MongoDB`. Ці типи такі:

`String`

`Номер`
`Date`
`Buffer`
`Boolean`
`mixed`
`ObjectId` (* унікальний ідентифікатор об’єкта, первинний ключ, _id)
`Array`

Для кожного типу даних можна:

- встановити значення за замовчуванням
- задати користувальницьку функцію перевірки даних
- вказати, що поле необхідно заповнити
- задати get-функцію (гетер), яка дозволяє вам проводити маніпуляції з даними до їх повернення як об’єкта
- встановити set-функцію (* сеттер), яка дозволяє вам проводити маніпуляції з даними до їх збереження в базу даних
- визначити індекси для швидшого отримання даних

Крім цих загальних можливостей для деяких типів даних також можна налаштувати особливості збереження та отримання даних із бази даних. Наприклад, для типу даних String можна вказати такі додаткові опції:

- конвертація даних у нижній регістр
- конвертація даних у верхній регістр
- обрізання даних перед збереженням
- визначення регулярного виразу, що дозволяє в процесі перевірки даних обмежити дозволені для збереження варіанти даних
- визначення переліку, який дозволяє встановити список допустимих рядків

Для властивостей типу `Number` і `Date` можна встановити мінімально і максимально допустиме значення.

Більшість із восьми допустимих типів даних повинні бути вам добре знайомі. Однак, деякі (`Buffer, Mixed, ObjectId і Array`) можуть спричинити труднощі.

Тип даних `Buffer` дозволяє зберігати двійкові дані. Типовим прикладом двійкових даних може бути зображення або закодований файл, наприклад, документ у PDF-форматі (* формат переносного документа).

Тип даних `Mixed` використовується для перетворення властивості на «нерозбірливе» поле (поле, в якому допустимі дані будь-якого типу). Подібно до того, як багато розробників використовують `MongoDB` для різних цілей, у цьому полі можна зберігати дані різного типу, оскільки відсутня певна структура. З обережністю використовуйте цей тип даних, оскільки він обмежує можливості `Mongoose`, наприклад, перевірку даних і відстеження змін сутності для автоматичного оновлення властивості при збереженні.

Тип даних ObjectId зазвичай використовується для визначення посилання на інший документ у вашій базі даних. Наприклад, якби у вас була колекція книг та авторів, документ книги міг би містити властивість ObjectId, яка посилається на певного автора документа.

Тип даних Array дозволяє зберігати JavaScript-подібні масиви. Завдяки цьому типу даних ви можете виконувати над даними типові JavaScript операції над масивами, наприклад, push, pop, shift, slice тощо.

Встановлення MongoDB

До того, як почати створювати схеми та моделі Mongoose, нам необхідно встановити та налаштувати MongoDB. Потрібно зайти на сторінку завантаження MongoDB (https://www.mongodb.com). Є кілька різних варіантів встановлення. виберіть Community Server.

Як тільки ви завантажили та встановили MongoDB для вибраної вами операційної системи, вам необхідно буде запустити базу даних.

Встановлення Mongoose

Mongoose – це бібліотека JavaScript. Використовуватимемо її в додатку Node.js.

Після переходу в консолі в папку, куди б ви хотіли встановити вашу програму, ви можете виконати наступні команди:

```
mkdir mongoose_basics
cd mongoose_basics
npm init
```

При ініціалізації програми залишайте значення всіх параметрів за промовчанням. Тепер встановлюємо модуль mongoose таким чином:

```
npm install mongoose –save
```

Після виконання всіх необхідних умов, підключаємося до бази даних MongoDB.

Розміщуємо код у файлі index.js:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/mongoose_basics');
```

У першому рядку коду ми підключаємо бібліотеку mongoose. Далі відкриваємо з'єднання з базою даних mongoose_basics, використовуючи функцію connect.

Функція connect приймає ще два інші необов'язкові параметри. Другий параметр - об'єкт опцій, де можна вказати, за потреби, наприклад, username (ім'я користувача) і password (пароль). Третій параметр, який також може бути другим, якщо у вас не

визначені опції, - це функція зворотного виклику, яка буде викликана після спроби з'єднання з базою даних. Функцію зворотного дзвінка можна використовувати двома способами:

```
    mongoose.connect(uri, options, function(error) {
    // Check error in initial connection. There is no 2nd param
to the callback.
    });
    // Or using promises
    mongoose.connect(uri, options).then(
    () => { /** ready to use. The `mongoose.connect()` promise
resolves to undefined. */ },
    err => { /** handle initial connection error */ }
    );
```

Щоб уникнути потенційної необхідності введення `JavaScript Promises`, модна використовувати перший спосіб.

Нижче наведено оновлений index.js:

```
    var mongoose = require('mongoose');
    mongoose.connect('mongodb://localhost/mongoose_basics',
function (err) {
       if (err) throw err;
       console.log('Successfully connected');
    });
```

У разі помилки при підключенні до бази даних викидається виняток, і подальше виконання функції переривається. При відсутності помилки консоль виводиться повідомлення про успішне з'єднання.

Тепер `Mongoose` встановлено та підключено до бази даних під назвою mongoose_basics

Визначення Mongoose Schema (*схеми)

Вище був об'єкт user, який мав дві властивості: firstName та lastName. У наступному прикладі можна побачити перероблений цей документ у схему:

```
    var userSchema = mongoose.Schema({
        firstName: String,
        lastName: String
    });
```

Це дуже проста схема, яка містить лише дві властивості без атрибутів, пов'язаних з нею. Давайте поширимо наш приклад, зробивши властивості first і last name дочірніми

об'єктами якості name. Властивість name міститиме властивості first і last name. Також додаємо властивість створеного типу Date.

```
var userSchema = mongoose.Schema({
    name: {
      firstName: String,
 lastName: String
    },
    created: Date
});
```

Як ви бачите, Mongoose дозволяє створювати дуже гнучкі схеми з безліччю можливих комбінацій організації даних.

У наступному прикладі створюємо дві нові схеми (author та book) та зв'язок з іншою схемою. Схема book міститиме посилання на схему author.

```
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
          firstName: String,
     lastName: String
 },
    biography: String,
    twitter: String,
    facebook: String,
    linkedin: String,
    profilePicture: Buffer,
    created: {
      type: Date,
      default: Date.now
    }
});
```

Вище наводиться схема author, яка поширює схему user, з попереднього прикладу. Щоб зв'язати Author і Book, у схемі author першою властивістю вказуємо _id типу ObjectId. _id - це стандартний синтаксис для позначення первинного ключа Mongoose і MongoDB. Далі, як і в схемі user, визначаємо властивість name, що містить first та last name автора.

Поширюючи схему user, схема author містить кілька додаткових властивостей типу String. Також додаємо властивість типу Buffer, в якому можна розмістити зображення профілю автора. Остання властивість містить дату створення автора; Зверніть увагу, що воно створене трохи інакше, тому що в ньому вказано значення за замовчуванням "зараз". При збереженні автора в базу даних, цій властивості буде надано значення поточної дати/часу.

Щоб завершити приклади схем, створіть схему book, яка містить посилання на автора за рахунок використання властивості типу ObjectId.

```
var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Author'
    },
    ratings: [
      {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
      }
    ],
    created: {
      type: Date,
      default: Date.now
    }
});
```

Схема book містить декілька властивостей типу String. Як згадувалося раніше, ця схема містить посилання схему author. Схема book також містить якість ratings типу Array, щоб продемонструвати вам можливості визначення схем. Кожен елемент цього масиву містить властивості summary, detail, numberOfStars і created date.

`Mongoose` дає вам можливість створювати схеми з посиланнями на інші схеми або, як у прикладі вище з властивістю ratings, дозволяє створювати `Array` дочірніх властивостей, який може міститися в прив'язаній схемі (author у нашому прикладі) або ж у поточній схемі, як у прикладі вище (схема book з властивістю ratings типу Array).

Створення та збереження Mongoose Models (*моделей)

Оскільки на прикладі схем author та book ми побачили гнучкість схеми `Mongoose`, продовжуємо використовувати їх та створити на їх основі моделі Author та Book.

```
var Author = mongoose.model('Author', authorSchema);
var Book = mongoose.model('Book', bookSchema);
```

Після збереження моделі в MongoDB створюється Document (* документ) з тими самими властивостями, що визначені у схемі, на основі якої було створено модель.

Щоб продемонструвати створення та збереження об'єкта, у наступному прикладі створюємо кілька об'єктів: одну модель Author та декілька моделей Book. Відразу після створення ці об'єкти будуть збережені в MongoDB за допомогою методу моделі save.

```
var jamieAuthor = new Author {
    _id: new mongoose.Types.ObjectId(),
    name: {
      firstName: 'James',
      lastName: 'Bond'
    },
    biography: 'Loren ipsimus',
    twitter: 'https://twitter.com/eldevelop',
    facebook: 'https://www.facebook.com/eldevelopcom/'
};
jamieAuthor.save(function(err) {
 if (err) throw err;
 console.log('Author successfully saved.');

 var mvcBook = new Book {
            _id: new mongoose.Types.ObjectId(),
            title: 'New book,
            author: jamieAuthor._id,
            ratings:[{
                summary: 'Great read'
            }]
 };

 mvcBook.save(function(err) {
      if (err) throw err;

      console.log('Book successfully saved.');
 });

 var knockoutBook = new Book {
            _id: new mongoose.Types.ObjectId(),
            title: 'Title new 1,
            author: jamieAuthor._id
 };

 knockoutBook.save(function(err) {
      if (err) throw err;

      console.log('Book successfully saved.');
 });
});
```

На початку прикладу ми створюємо та зберігаємо jamieObject, створений за допомогою моделі Author. У разі помилки всередині функції save об'єкта jamieObject програма викине виняток. У разі відсутності помилки всередині функції save будуть створені та збережені два об'єкти book. Подібно до об'єкта jamieObject, у цих об'єктах у разі виникнення помилки при збереженні викидається виняток. В іншому випадку в консоль виводиться повідомлення про успішне збереження.

Для створення посилання на Author обидва об'єкти book посилаються на первинний ключ _id схеми author у властивості author схеми book.

Перевірка даних перед збереженням

Загальноприйнято заповнення даних для створення моделі у формі на веб-сторінці. З цієї причини, добре б перевірити ці дані перед збереженням моделі `MongoDB`.

У наступному прикладі оновлюємо попередню схему author, додавши перевірку даних наступних властивостей: firstName, twitter, facebook та linkedin.

```
    var authorSchema = mongoose.Schema({
        _id: mongoose.Schema.Types.ObjectId,
        name: {
          firstName: {
               type: String,
               required: true
          },
          lastName: String
     },
     biography: String,
     twitter: {
          type: String,
          validate: {
               validator: function(text) {
                    return   text.indexOf('https://twitter.com/')
=== 0;
               },
               message:   'Twitter   handle   must   start   with
https://twitter.com/'
          }
     },
     facebook: {
          type: String,
          validate: {
               validator: function(text) {
                    return
text.indexOf('https://www.facebook.com/') === 0;
               },
```

```
                message:     'Facebook     must     start     with
https://www.facebook.com/'
           }
      },
      linkedin: {
           type: String,
           validate: {
                validator: function(text) {
                     return
text.indexOf('https://www.linkedin.com/') === 0;
                },
                message:     'LinkedIn     must     start     with
https://www.linkedin.com/'
           }
      },
      profilePicture: Buffer,
      created: {
           type: Date,
           default: Date.now
      }
     });
```

Для якості першоїназви був заданий атрибут, що вимагається. Тепер, коли виклик функції save, `Mongoose` поверне помилку з повідомленням про необхідність вказати значення властивості firstName. Робимо властивість lastName без необхідності вказівки його значення на випадок, якщо авторами в базі даних були Cher або Madonna (* відсутнє прізвище).

Для властивостей twitter, facebook і linkedin використовуються подібні валідатори. Вони перевіряються на відповідність початку їх значень відповідного доменного імені соціальних мереж. Оскільки це є необов'язковими для заповнення поля, валідатор застосовується лише у разі надходження даних для цієї властивості.

Пошук та оновлення даних

Введення в Mongoose не було б завершеним без прикладу пошуку запису та оновлення однієї або більше властивостей цього об'єкта.

Mongoose пропонує кілька різних функцій для пошуку даних певної моделі. Ці функції наступні: find, findOne і findById.

Функції find і findOne отримують як аргумент об'єкт, що дозволяє здійснювати складні запити. Функція findById отримує лише одне значення функції зворотного виклику (скоро буде приклад). У наступному прикладі можна побачити, як можна зробити вибірку книг, що містять у своїй назві рядок 'mvc'.

```
Book.find({
 title: /mvc/i
```

```
}).exec(function(err, books) {
 if (err) throw err;

 console.log(books);
});
```

Усередині функції find робимо пошук нечутливого до регістру рядка 'mvc' за якістю title. Це здійснюється за допомогою того ж синтаксису, що використовується для пошуку рядка `JavaScript`.

Функцію find також можна "причепити" до інших методів запиту, наприклад, where, and, or, limit, sort, any і т.д.

Давайте поширимо наш попередній приклад, обмеживши кількість результатів до п'яти перших книг і відсортувавши їх за датою створення за спаданням. Результатом будуть перші п'ять нових книг, що містять у назві рядок 'mvc'.

```
Book.find({
 title: /mvc/i
}).sort('-created')
.limit(5)
.exec(function(err, books) {
 if (err) throw err;

 console.log(books);
});
```

Після застосування функції find порядок наступних функцій немає значення, оскільки з усіх зчеплених функцій формується єдиний запит і функції не виконуються до виклику функції exec.

Функція findById виконується трохи інакше. Вона виконується відразу ж і приймає як один з аргументів функцію зворотного виклику, і не дозволяє зчеплення функцій. У наступному прикладі запитаємо необхідного автора з його _id.

```
    Author.findById('59b31406beefa1082819e72f',     function(err,
author) {
        if (err) throw err;

        console.log(author);
    });
```

У вас значення _id може бути трохи іншим. Копіюємо значення _id з попереднього console.log, коли здійснювали пошук книг, що містять у назві рядок 'mvc'.

Відразу після повернення об'єкта ви можете змінити будь-яку з його властивостей та оновити його. Як тільки ви внесли необхідні зміни, ви викликаєте метод save також, як ви

робили при створенні об'єкта. У наступному прикладі поширимо приклад з функцією findbyId і оновлюємо властивість linkedin автора.

```
    Author.findById('59b31406beefa1082819e72f',     function(err,
author) {
     if (err) throw err;

     author.linkedin  =  'https://www.linkedin.com/in/jamie-munro-
8064ba1a/';

     author.save(function(err) {
          if (err) throw err;

          console.log('Author updated successfully');
     });
    });
```

Після успішного отримання автора встановлюється значення властивості linkedin і викликається функція save. Mongoose здатна помітити зміну властивості linkedin і передати стан, оновлений тільки за модифікованими властивостями, в MongoDB. У разі виникнення помилки при збереженні буде викинуто виняток і програма припинить роботу. За відсутності помилок у консоль буде виведено повідомлення про успішну зміну.

Також Mongoose надає можливість знайти об'єкт та одразу оновити його за допомогою функцій з відповідними назвами: findByIdAndUpdate та findOneAndUpdate. Давайте оновимо попередній приклад, щоб показати функцію findByIdAndUpdate у дії.

```
    Author.findByIdAndUpdate('59b31406beefa1082819e72f',
        {   linkedin:   'https://www.linkedin.com/in/jamie-munro-
8064ba1a/' },
        function(err, author) {
         if (err) throw err;

         console.log(author);
    });
```

У попередньому прикладі властивості, які ми хочемо оновити, передаються у функцію findByIdAndUpdate як об'єкт другим параметром. Функція зворотного дзвінка є третім параметром. Після вдалого оновлення повернутий об'єкт author містить оновлену інформацію. Він виводитиметься в консоль, щоб ми побачили оновлені властивості автора.

Повний код прикладу

Ми створили два додаткові файли: author.js та book.js. Дані файли містять відповідні редеління схем та створення моделей. Останній рядок коду робить модель доступною для використання у файлі index.js.

файл author.js:

```
var mongoose = require('mongoose');
var authorSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    name: {
      firstName: {
           type: String,
           required: true
      },
      lastName: String
 },
 biography: String,
 twitter: {
      type: String,
      validate: {
           validator: function(text) {
                return   text.indexOf('https://twitter.com/')
=== 0;
           },
           message:   'Twitter   handle   must   start   with
https://twitter.com/'
      }
 },
 facebook: {
      type: String,
      validate: {
           validator: function(text) {
                return
text.indexOf('https://www.facebook.com/') === 0;
           },
           message:    'Facebook    must    start    with
https://www.facebook.com/'
      }
 },
 linkedin: {
      type: String,
      validate: {
           validator: function(text) {
                return
text.indexOf('https://www.linkedin.com/') === 0;
           },
           message:    'LinkedIn    must    start    with
https://www.linkedin.com/'
      }
 },
 profilePicture: Buffer,
 created: {
      type: Date,
```

```
        default: Date.now
    }
});
var Author = mongoose.model('Author', authorSchema);
module.exports = Author;
```

файл book.js:

```
var mongoose = require('mongoose');
var bookSchema = mongoose.Schema({
    _id: mongoose.Schema.Types.ObjectId,
    title: String,
    summary: String,
    isbn: String,
    thumbnail: Buffer,
    author: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Author'
    },
    ratings: [
        {
            summary: String,
            detail: String,
            numberOfStars: Number,
            created: {
                type: Date,
                default: Date.now
            }
        }
    ],
    created: {
        type: Date,
        default: Date.now
    }
});
var Book = mongoose.model('Book', bookSchema);
module.exports = Book;
```

файл index.js:

```
var mongoose = require('mongoose');
var Author = require('./author');
var Book = require('./book');
mongoose.connect('mongodb://localhost/mongoose_basics',
function (err) {
    if (err) throw err;

    console.log('Successfully connected');
```

```
     var jamieAuthor = new Author({
          _id: new mongoose.Types.ObjectId(),
          name: {
               firstName: 'Jamie',
               lastName: 'Munro'
          },
          biography: 'Jamie is the author of ASP.NET MVC 5 with
Bootstrap and Knockout.js.',
          twitter: 'https://twitter.com/endyourif',
          facebook:        'https://www.facebook.com/End-Your-If-
194251957252562/'
     });
     jamieAuthor.save(function(err) {
          if (err) throw err;

          console.log('Author successfully saved.');

          var mvcBook = new Book({
               _id: new mongoose.Types.ObjectId(),
               title:   'ASP.NET   MVC   5   with   Bootstrap   and
Knockout.js',
               author: jamieAuthor._id,
               ratings:[{
                    summary: 'Great read'
               }]
          });

          mvcBook.save(function(err) {
               if (err) throw err;

               console.log('Book successfully saved.');
          });

          var knockoutBook = new Book({
               _id: new mongoose.Types.ObjectId(),
               title: 'Knockout.js: Building Dynamic Client-Side
Web Applications',
               author: jamieAuthor._id
          });

          knockoutBook.save(function(err) {
               if (err) throw err;

               console.log('Book successfully saved.');
          });
     });
    });
```

У наведеному вище прикладі всі дії Mongoose містяться всередині функції connect. Файли author і book підключаються за допомогою функції require після підключення mongoose.

Якщо MongoDB запущено, ви тепер можете запустити повну програму `Node.js` за допомогою наступної команди:

```
node index.js
```

Після збереження деяких даних у базу оновлюємо файл index.js, додавши функції пошуку таким чином:

```
var mongoose = require('mongoose');
var Author = require('./author');
var Book = require('./book');
mongoose.connect('mongodb://localhost/mongoose_basics',
function (err) {
    if (err) throw err;

 console.log('Successfully connected');

 Book.find({
     title: /mvc/i
 }).sort('-created')
 .limit(5)
 .exec(function(err, books) {
     if (err) throw err;

     console.log(books);
 });

 Author.findById('59b31406beefa1082819e72f',    function(err,
author) {
     if (err) throw err;

     author.linkedin  =  'https://www.linkedin.com/in/jamie-
munro-8064ba1a/';

     author.save(function(err) {
         if (err) throw err;

         console.log('Author updated successfully');
     });
 });

 Author.findByIdAndUpdate('59b31406beefa1082819e72f',       {
linkedin:  'https://www.linkedin.com/in/jamie-munro-8064ba1a/'  },
function(err, author) {
```

```
        if (err) throw err;

        console.log(author);
  });
});
```

Можна запустити скрипт за допомогою наступної команди:

```
node index.js
```

Завдання до лабораторної роботи № 6:

Спроектувати базу даних з таблицею студентів яка має містити дані про студента відповідно до наданої таблиці у макеті **Messages**
https://cacoo.com/diagrams/ZvVhYS3UpG5PdbBy/EDE3A

Обираємо для розробки шаблон Messages

Розробити 2 складові частини проекту :

- Створюємо ДБ тут https://www.mongodb.com
- Використати бібліотеку socket.io
- Сайт з формою, що надсилає на сервер дані чату
- Серверна частина перевіряє чи повідомлення конкретному студенту чи декільком/всім студентам і повертає відповідь (response) всім кому це повідомлення адресовано
- Серверна частина має зберегти повідомлення у бд щоб в подальшому показати їх користувачам
- Сайт реагує на відповіді. І в потрібних чатах показує повідомлення