MongoDB Assignment

Introduction

This assignment will provide an opportunity to practice using MongoDB, a non relational database. We will use MongoDB to permanently store student attempts at answering questions on quizzes stored in the database. Mongoose will be used to create schemas and models for the quizzes, questions, and attempts. Attempts will be scored and reported for each of the quizzes.

Populate a MongoDB instance

In the previous assignment we used arrays in files to represent data we accessed through a RESTful API. In this assignment we are going to replace the arrays with a MongoDB instance. We'll populate the database with the same data we used in the previous assignment, and then use the data in the database. Start the MongoDB server and connect to it with the mongo client. Create a database called whiteboard where we will store the data. In the whiteboard database, create two collections called quizzes and questions. Use the following data to create documents in the quizzes collection.

quizzes collection

```
{"_id": "123", "title": "Quiz 1"}
{"_id": "234", "title": "Quiz 2"}
{"_id": "345", "title": "Quiz 3"}
```

Then create a collection called questions that contains the following documents. Note that the quizId properties in the questions have the same values as some of the _id properties in the quizzes collection.

questions collection

```
    {"_id": "123", "title": "JPA True False", "quizId": "234", "question":"JPA stands for Java Persistence API", "correct": "true", "type": "TRUE_FALSE"}

     {"_id": "234", "title": "JPA Multiple Choice", "quizId": "234", "question": "What does JPA stand for?", "correct": "Java Persistence API", "type": "MULTIPLE_CHOICE", "choices": ["Java Pseudo API", "Java Persistence API", "JSON Persistence API", "JavaScript Persistence API"]}

     {"_id": "345", "title": "JSON True False", "quizId": "234", "question": "JSON stands for Java Object Notation", "correct": "false","type": "TRUE_FALSE"}

    {"_id": "456", "title": "JSON Multiple Choice", "quizId": "345", "question": "What does JSON stand for?", "correct": "JavaScript Object Notation", "type": "MULTIPLE_CHOICE", "choices": ["JavaScript Object Notation", "JavaScript Object Normalization", "Java Object Normalization", "Java Object Notation"]}

   {"_id": "567", "title": "CSS True False", "quizId": "345", "question":
```

```
"CSS stands for Cascading Style Sheet", "correct": "true", "type": "TRUE_FALSE"}

   {"_id": "678", "title": "CSS Multiple Choice", "quizId": "123", "question":
"What does CSS stand for?", "correct": "Cascading Style Sheet", "type":
"MULTIPLE_CHOICE", "choices": ["Class Style Sheet", "Cascading Screen Style",
"Cascading Style Sheet", "Cascading Style Screen"]}
```

Create the Schemas

Now we will create mongoose schema files that represent the structure of the documents in the questions and quizzes collections. We'll start with the questions and then move on to the quizzes. In a file called questions-schema.js in directory **`models/questions`**, create a Mongoose schema that represents the questions in a quiz. Use the example below as a guide. Note that we are defining the datatype of the field _id. This means that we are making ourselves responsibles for the values of the primary keys, instead of letting mongo choose the unique value for these fields. The reason we are doing this is that we are populating the database with a predetermined set of documents that are related to one another.

`models/questions/questions-schema.js`

```
const mongoose = require('mongoose')
const questionsSchema = mongoose.Schema({
  _id: String,
  title: String,
  question: String,
  correct: String,
  answer: String,
  type: {type: String, enum: ['TRUE_FALSE', 'MULTIPLE_CHOICE', 'FILL_BLANKS']},
  choices: [String]
}, {collection: 'questions'})
module.exports = questionsSchema
```

Now let's create the schema for the quizzes in the following file: **`models/quizzes/quizzes-schema.js`**. Create a Mongoose schema that represents the quizzes as a collection of questions. User the following as an example. Note that here too we are declaring the _id field. Also note that the quizzes schema contains an array of question references we can then use to dereference and retrieve the actual questions for the quiz.

`models/quizzes/quizzes-schema.js`

```
const mongoose = require('mongoose')
const quizzesSchema = mongoose.Schema({
  _id: String,
  title: String,
  questions: [{
     type: String,
     ref: 'QuestionsModel'
  }]
```

```
}, {collection: 'quizzes'})
module.exports = quizzesSchema
```

Next we are going to need a collection where we can store the answers students give when taking a test. We will let students take the quizzes as many times as they want. Each time they answer the questions of a quiz, we will record it as a separate attempt. Let's call the collection quiz-attempts. In **`models/quiz-attempts`** create a mongoose schema file called **`quiz-attempts-schema.js`** that represents a student's attempt to answer the questions on a quiz. The attempts will be a record of the answers students gave for each of the questions. We'll store these as an array of questions containing the answers students gave for each question. Use the following as an example

`models/quiz-attempts/quiz-attempts-schema.js`

```
const mongoose = require('mongoose')
const questionSchema = require('../questions/questions-schema')
const quizAttempts = mongoose.Schema({
  _id: String,
  score: Number,
  quiz: {type: String, ref: 'QuizzesModel'},
  answers: [questionSchema]
}, {collection: 'quizAttempts'})
module.exports = quizAttempts
```

Create the Mongoose Models

Mongoose shemas define the structure of the data. Mongoose models provide the operations for actually storing and retrieving data to/from the database. In **`models/questions/questions-model.js`**, create a Mongoose model for CRUDing questions in MongoDB. Use the following as an example

`models/questions/questions-model.js`

```
const mongoose = require('mongoose')
const questionsSchema = require('./questions-schema')
const questionsModel = mongoose.model(
  'QuestionsModel',
  questionsSchema
)
module.exports = questionsModel
```

In **`models/quizzes/quizzes-model.js`**, create a Mongoose model for CRUDing quizzes in MongoDB. Use the following as an example

`models/quizzes/quizzes-model.js`

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

```
const quizSchema = require('./quizzes-schema')
const quizModel = mongoose.model(
  'QuizModel',
  quizSchema
)
module.exports = quizModel
```

In **`models/quiz-attempts/quiz-attempts-model.js`**, create a Mongoose model for CRUDing quiz attempts in MongoDB. Use the following as an example

`models/quiz-attempts/quiz-attempts-model.js`

```
const mongoose = require('mongoose')
const quizAttemptSchema = require('./quiz-attempts-schema')
const quizAttemptModel = mongoose.model(
  'QuizAttemptModel',
  quizAttemptSchema
)
module.exports = quizAttemptModel
```

Create the Data Access Objects (DAOs)

Data Access Objects are design patterns that encapsulate data access operations into modular, reusable interfaces that shield the rest of the application from the details and complexities of interacting with a database. In **`daos/questions-dao.js`**, create a DAO to CRUD questions in MongoDBB. Use the following as an example. Feel free to modify as necessary.

`daos/questions-dao.js`

```
const questionsModel = require('../models/questions/questions-model')
const quizzesModel = require('../models/quizzes/quizzes-model')

const findAllQuestions = () => questionsModel.find()
const findQuestionById = (qid) => questionsModel.findById(qid)
const findQuestionsForQuiz = (qzid) => quizzesModel.findById(qzid)
    .populate('questions').then(quiz => quiz.questions)
module.exports = { findAllQuestions, findQuestionById, findQuestionsForQuiz }
```

In **`daos/quizzes-dao.js`**, create a DAO to CRUD quizzes in MongoDB. Use the following as an example. Feel free to modify as necessary.

`daos/quizzes-dao.js`

```
const quizzesModel = require('../models/quizzes/quizzes-model')
const findAllQuizzes = () => quizzesModel.find()
```

```
const findQuizById = (quizId) => quizzesModel.findById(quizId)
module.exports = { findAllQuizzes, findQuizById }
```

In **daos/quiz-attempts-dao.js** create a DAO to CRUD quiz attempts in MongoDB. Use the following as an example. Note that scoreQuiz function below computes the score of a quiz by adding up how many correct answers a student got compared to the total number of questions. Feel free to modify as necessary.

daos/quiz-attempts-dao.js

```
const quizAttemptsModel = require('../models/quiz-attempts/quiz-attempts-model')

const scoreQuiz = (questions) => {
  let numberOfCorrectQuestions = 0
  questions.forEach(question => question.answer === question.correct ?
        numberOfCorrectQuestions++ : numberOfCorrectQuestions)
  return 100 * numberOfCorrectQuestions / questions.length }

const findAttemptsForQuiz = (qzid) => quizAttemptsModel.find({quiz: qzid}).populate('quiz', 'title _id')
const createAttempt = (qid, attempt) =>
  quizAttemptsModel.create({ quiz: qid, answers: attempt, score: scoreQuiz(attempt) })

module.exports = { createAttempt, findAttemptsForQuiz }
```

Refactor Services to Use DAOs Instead of JSON Arrays

The previous implementation of the services was using arrays to store the data. We need to refactor the services to use the data se instead. Instead of returning arrays and values, they will return promises from the mongoose models responsible for communicating with the database. Refactor **questions-service.js** so that now it uses the new DAO. Use the following as an example. Feel free to modify as necessary.

services/questions-service.js

```
const questionsDao = require('../daos/questions-dao')
const findAllQuestions = () => questionsDao.findAllQuestions()
const findQuestionById = (qid) => questionsDao.findQuestionById(qid)
const findQuestionsForQuiz = (qid) => questionsDao.findQuestionsForQuiz(qid)
module.exports = { findAllQuestions, findQuestionById, findQuestionsForQuiz }
```

Refactor **quizzes-service.js** so that now it uses the new DAO. Use the following as an example. Feel free to modify as necessary.

services/quizzes-service.js

```
const quizzesDao = require('../daos/quizzes-dao')
const findAllQuizzes = () => quizzesDao.findAllQuizzes()
```

```
const findQuizById = (qid) => quizzesDao.findQuizById(qid)
module.exports = { findAllQuizzes, findQuizById }
```

Refactor Controllers to Use Asynchronous Functions

The previous implementation of the controllers used a service that was working with arrays. Since the services now work with a database instead, the controllers need to be refactored to use the new implementation. Services now return promises instead of actual data. Therefore controllers must register callbacks to handle the asynchronous responses from the database. Refactor **`questions-controller.js`** to use an asynchronous service API. Use the following as an example. Feel free to modify as necessary.

`controllers/questions-controller.js`

```
const questionsService = require('../services/questions-service')
module.exports = function(app) {
  app.get('/api/quizzes/:qid/questions', (req, res) =>
    questionsService.findQuestionsForQuiz(req.params['qid'])
      .then(questions => res.json(questions)))
  app.get('/api/questions', (req, res) =>
    questionsService.findAllQuestions()
      .then(allQuestions => res.json(allQuestions)))
  app.get('/api/questions/:qid', (req, res) =>
    questionsService.findQuestionById(req.params['qid'])
      .then(question => res.json(question)))
}
```

Refactor **`quizzes-controller.js`** to use an asynchronous service API. Use the following as an example. Feel free to modify as necessary.

`controllers/quizzes-controller.js`

```
const quizzesService = require('../services/quizzes-services')
module.exports = function (app) {
  app.get('/api/quizzes', (req, res) =>
    quizzesService.findAllQuizzes()
      .then(allQuizzes => res.json(allQuizzes)))
  app.get('/api/quizzes/:qzid', (req, res) =>
    quizzesService.findQuizById(req.params['qzid'])
      .then(quiz => res.json(quiz)))
}
```

Implement **`quiz-attempts-controller.js`** to use the quiz attempts DAO. Use the following as an example. Feel free to modify as necessary.

`controllers/quiz-attempts-controller.js`

```
const quizAttemptDao = require('../daos/quiz-attempts-dao')
module.exports = (app) => {
  app.post('/api/quizzes/:qid/attempts', (req, res) =>
    quizAttemptDao.createAttempt( req.params.qid, req.body)
      .then(attempt => res.send(attempt)))
  app.get('/api/quizzes/:qid/attempts', (req, res) =>
    quizAttemptDao.findAttemptsForQuiz(req.params.qid)
      .then(attempts => res.send(attempts)))
}
```

Register the controllers in **`server.js`**. Use the following as an example. Feel free to modify as necessary.

`server.js`

```
var bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
require('./controllers/quizzes-controller')(app)
require('./controllers/questions-controller')(app)
require('./controllers/quiz-attempts-controller')(app)
```

Refactor React Components

Refactor the React components implemented in the previous assignment to list the attempts for each of the quizzes and the score for each of the attempts. Replace the Grade button with a Submit button that calls a new function called ***submitQuiz()***. The ***submitQuiz()*** should post the quiz to the attempts RESTful Web service endpoint. Use the following as an example. Feel free to modify as necessary.

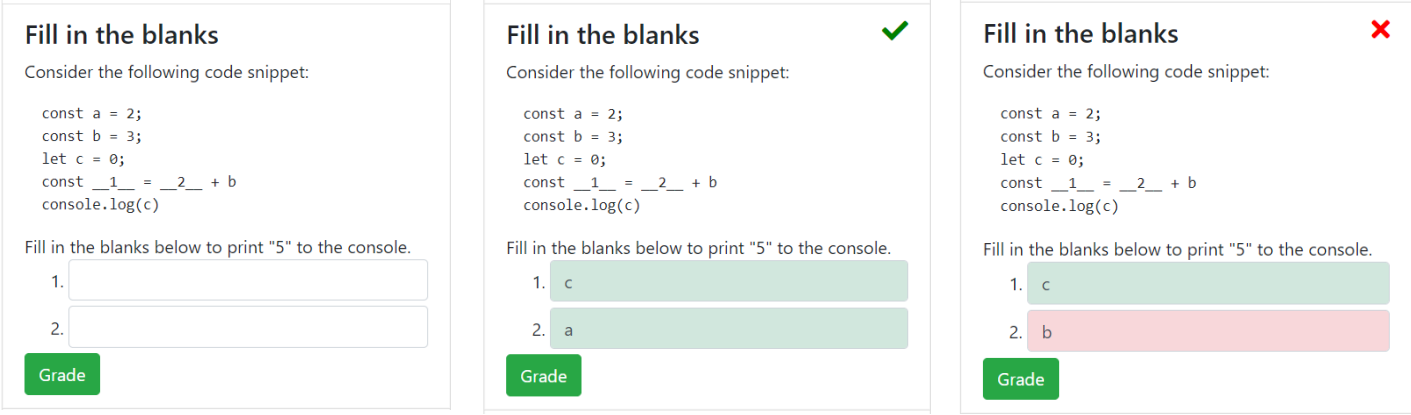
`services/quizzes-service.js`

```
submitQuiz = (quizId, questions) => {
 fetch(`http://localhost:3000/api/quizzes/${quizId}/attempts`, {
   method: 'POST',
   body: JSON.stringify(questions),
   headers: {
     'content-type': 'application/json'
   }
 }).then(response => response.json())
   .then(result => console.log(result))
}
```

New Fill Blanks Question Component (Optional, won't be graded)

To practice all you've learned so far, on your own, create a new type of question that implements fill in the blanks. The question should render a question and several blanks students can fill in as illustrated below. Each blank has a correct answer and the score of the question is based on the how many student answers match

the correct answers. The question editor should allow to enter the question, how many blanks, and the correct answers for each of the blanks. Use the wireframes below as a guide of how the user interface should look like. This question will not be graded and is only provided for you to practice.



Deliverables

As a deliverable deploy the React and Node.js applications to Heroku. When deployed on Heroku, the Node.js server should be using a remote MongoDB database hosted on a service such as mLab or ObjectRocket. Provide all Git repositories and application URLs on Heroku.