Lab 3: Interacting with Databases

Learning Objectives

- Practice validating data using Joi
- Implement a SQLite database for persisting data
- Practice structuring your applications in adherence to the MVC pattern

This week's exercises

The exercises today build on top of Exercise 4 and 5 from last week. Make sure that you have first implemented your own version of the ShoppingCartAPI before starting on this week's lab.

METHOD	ROUTE	REQUEST	RESPONSE	DESCRIPTION
GET	/cart	-	[cart_items]	Returns a list of all items in the cart
POST	/cart	item_name, item_price, quantity	item_id, item_name, item_price, quantity	Adds a new item to the cart
GET	/cart/:id	-	item_id, item_name, item_price, quantity	Gets a single item from the cart
PATCH	/cart/:id	quantity	item_id, item_name, item_price, quantity	Updates a single item within the list (users can only update the quantity of items)
DELETE	/cart/:id	-	"<Item name> Deleted"	Deletes an item from the list

Exercise 1: Validation with Joi

The ShoppingCartAPI contains two endpoints that handle incoming data. This data needs to be validated so that we can be sure it adheres to the specification.

Using the Joi library, validate all incoming data across your API. Use this week's lecture slides and Joi documentation (https://joi.dev/api/?v=17.6.1) to help you.

Exercise 2: Persistence

Using the slides from this week's lecture to help you, change your ShoppingCartAPI so that data is persisted to a SQLite DB rather than a single array.

I know that there's not a lot of instruction here – that is intentional. This exercise forces you to go through the lecture slides and fully understand the code in order for you to be able to use it in your own application.

Exercise 3: Refactor

Using the slides from this week's lecture to help you, refactor your API so that it conforms to the MVC directory structure.

1. Take a backup of your codebase beforehand
2. Start by creating the directory structure
3. Using the slides to help you, create the routes file – make sure you import the routes file in your project
4. Using the slides to help you, create the model's files and functions
5. Link the routes to the models by implementing the controller functions

This exercise is difficult. It is always easier to write clean code from the day one rather than refactoring messy code later down the line. However, it is an important skill that you will inevitably have to do regularly in larger projects. Doing this exercise should also make the next exercise easy – where we start writing some assignment code.

Exercise 4: Assignment!

Exercise 4.1: Starter code

If you have not done so already, download and run the starter code. Instructions for doing so are in the repository's README: https://github.com/ash-williams/fsd_blog_engine_server

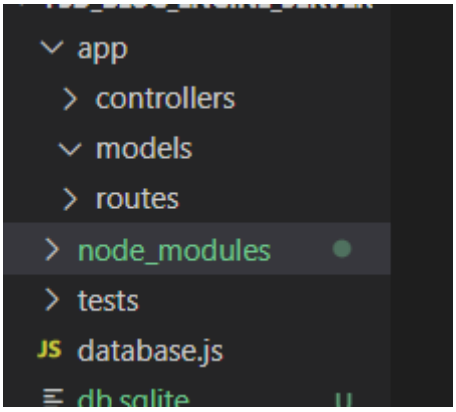
At this point, most tests will fail – because we have not yet implemented anything.

Open the directory in VSCode ready to code.

Note: All your code should be written inside of the app directory. There is no need to edit any of the other files (unless instructed to)

Exercise 4.2: Directory structure

1. In your app directory, create directories for storing your routes, models, and controllers
2. You can delete the .txt file that is currently in there.



Exercise 4.3: Routes file

1. In your routes directory, create a file called "articles.routes.js"
2. In this file, we want to create the routes for each of the five article endpoints. However, we're not going to implement the endpoints here.

```
1
2  module.exports = function(app){
3
4      app.route("/articles")
5          .get()
6          .post();
7
8      app.route("/articles/:article_id")
9          .get()
10         .patch()
11         .delete();
12 }
```

3. Instead, import your controller and call functions for each of the endpoints (we will implement these next)

```
const articles = require("../controllers/articles.controllers");

module.exports = function(app){

    app.route("/articles")
        .get(articles.getAll)
        .post(articles.create);

    app.route("/articles/:article_id")
        .get(articles.getOne)
        .patch(articles.updateArticle)
        .delete(articles.deleteArticle);
}
```

4. In your controller's directory, create a file called "articles.controllers.js"
5. Create each of the functions that you have called your routes file, for now just set each one to return a 500 response to the client.

```
const getAll = (req, res) => {
    return res.sendStatus(500);
}

const create = (req, res) => {
    return res.sendStatus(500);
}

const getOne = (req, res) => {
    return res.sendStatus(500);
}

const updateArticle = (req, res) => {
    return res.sendStatus(500);
}

const deleteArticle = (req, res) => {
    return res.sendStatus(500);
}
```

6. At the bottom of the controller, export the functions so that they can be accessed by other files (e.g., the routes file)

```
module.exports = {
    getAll: getAll,
    create: create,
    getOne: getOne,
    updateArticle: updateArticle,
    deleteArticle: deleteArticle
}
```

7. At the moment, your routes file isn't linked to your main application (i.e., the server.js file). Link them by adding the following line to the server.js file (linking routes files to the server.js file is the only time you should write code that isn't in the app directory)

```
// Other API endpoints: Links go here...
require("./app/routes/articles.routes")(app);
```

In this line, you pull in the article routes, but pass the express app to the routes file so that we can use it.

8. Run the server and test using Postman. All routes should return a 500 response because they have not yet been implemented.

Exercise 4.4: Getting all articles

1. In the models directory, create a file called “articles.models.js” - in this file, we will create the functions that interact with the database.
2. At the top of the models file, require in the database file

```
const db = require("../../database");
```

3. Create a function called getAllArticles. This function will pull all the articles from the database and return them as an array.
   Note: This function is going to take in a function called “done” as a parameter. Once we have finished processing the articles, we will execute the done function. This is called a callback function and allows the code calling the function to define what happens when getAllArticles function has finished. We’ll see how this works shortly.

```
3  const getAllArticles = (done) => {
4
5  }
```

4. For getting all of the articles from the database, we can use the db.each() function. Remember from the slides that the each() function takes in four parameters:
   a. The SQL statement to execute
   b. Any parameters needed for the SQL statement
   c. A function that will execute for each record
   d. A function that will execute once every record has been processed.

   For this function, each record will be added to an array. Once complete, the array will be passed to the done function so that the code calling the getAllArticles function can determine what to do with them.
   Note: now is our opportunity to make sure the array of results adheres to the API specification (meaning that the controller doesn’t need to reformat the array). We also convert all dates from the integers that are stored in the database, to a nice date format.

```
const getAllArticles = (done) => {
    const results = [];

    db.each(
        "SELECT * FROM articles",
        [],
        (err, row) => {
            if(err) console.log("Something went wrong: " + err);

            results.push({
                article_id: row.article_id,
                title: row.title,
                author: row.author,
                date_published: new Date(row.date_published).toLocaleDateString(),
                date_edited: new Date(row.date_edited).toLocaleDateString(),
                article_text: row.article_text
            });
        },
        (err, num_rows) => {
            return done(err, num_rows, results);
        }
    )
}
```

5. Export the function so that it can be seen by other files (i.e., the controller)

```
module.exports = {
    getAllArticles: getAllArticles
}
```

6. In the article controller, require in the model file so that we can access the functions.

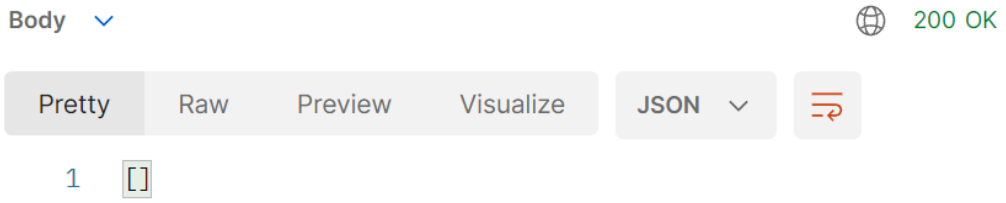
```
const articles = require("../models/articles.models");
```

7. In the getAll function, we are going to call the getAllArticles function that we've just created. We need to define the callback function (the done parameter) so that we can define how the response is handled. This this case, we are going to check that there is no error, and then return the results to the client.

```
const getAll = (req, res) => {
    articles.getAllArticles((err, num_rows, results) => {
        if(err) return res.sendStatus(500);

        return res.status(200).send(results);
    })
}
```

8. Save and test. You should get an empty list returned to the client now (because there are no articles in our database)



Exercise 4.5: Creating a new article

We have now implemented our first assignment endpoint, and if you run the tests you'll see that there are less failed tests then previously – which means you already have some marks in the bag 😊

For this second endpoint, we are going to implement the endpoint to add new articles. The process is like before, only now we are inserting data into the database.

1. In the models file, create a new function called addNewArticle. Make sure you export it in the module exports so that it can be accessed by other files. This function will contain two parameters: the article object to be added to the database, and a callback so that the calling code can specify what happens afterwards.

```
const addNewArticle = (article, done) => {

}

module.exports = {
    getAllArticles: getAllArticles,
    addNewArticle: addNewArticle
}
```

2. To execute a SQL INSERT statement, we can use db.run(). The run() function takes in three parameters:
    a. The SQL statement to execute
    b. An array of parameters for the SQL statement
    c. A function to execute on completion

    Important bits to note:
    - We can access the current date/time in JS using Date.now()
    - The API specification tells us what data will be sent by the client, and the assessment specification shows what tables and columns there are in the database, so the run() function just needs to map the two together.
    - Eventually we will need to come back to this function. The database requires us to state who created the article (created_by). Once we can implement authentication, we will be able to extract this automatically based on the ID of the person who is logged in. For now, we will just default this value to 1
    - The run() function allows us to access this.lastID for accessing the primary key of the record that has just been added to the database. This is useful for being able to return the article ID to the client

```
const addNewArticle = (article, done) => {

    let date = Date.now();
    const sql = 'INSERT INTO articles (title, author, date_published, date_edited, article_text, created_by) VALUES (?,?,?,?,?,?)';
    let values = [article.title, article.author, date, date, article.article_text, 1];

    db.run(
        sql,
        values,
        function(err){
            if(err) return done(err, null);

            return done(null, this.lastID);
        }
    )
}
```

3. In the controller, edit the create function so that it now gets the article data out of req.body, and sends it to the model function that we have just created.

```
const create = (req, res) => {
    let article = Object.assign({}, req.body);

    articles.addNewArticle(article, (err, id) => {
        if(err) return res.sendStatus(500);

        return res.status(201).send({article_id: id})
    })
}
```

4. But what if the client sends some malformed data? Use Joi to validate the incoming data against the specification. This needs to be done prior to calling the addNewArticle function and you should return a 400 Bad Request if the data doesn't validate.

```
const create = (req, res) => {

    const schema = Joi.object({
        "title": Joi.string().required(),
        "author": Joi.string().required(),
        "article_text": Joi.string().required()
    })

    const { error } = schema.validate(req.body);
    if(error) return res.status(400).send(error.details[0].message);

    let article = Object.assign({}, req.body);

    articles.addNewArticle(article, (err, id) => {
        if(err) return res.sendStatus(500);

        return res.status(201).send({article_id: id})
    })
}
```

5. Save and test your endpoint – your new articles should also now appear in the GET /articles list. Also, running npm test now shows that even less tests are failing – more marks!


```
[
    {
        "article_id": 1,
        "title": "5 Best Blog Engines",
        "author": "Megan",
        "date_published": "09/10/2022",
        "date_edited": "09/10/2022",
        "article_text": "This is the articles main body text."
    }
]
```


Exercise 4.6: Getting a single article

Things start getting a bit more straight forward now. We start the endpoints by creating the model function, and then we link up the route through the controller and add in any validation that we need.

1. For SELECT statements that return a single result, we can use db.get(). The get() function takes in three parameters:
    a. The SQL statement to execute
    b. Any parameters needed for the SQL statement
    c. A function to execute on completion

    Important bits to note:

    - If no row is found, we will send the digits 404 to the calling function so that they know that the row couldn't be found
    - If we find the row successfully, we structure the object as the API states so that we don't need to reformat the object in the controller
    - Don't forget to add the function to the module exports

```
const getSingleArticle = (id, done) => {
    const sql = 'SELECT * FROM articles WHERE article_id=?'

    db.get(sql, [id], (err, row) => {
        if (err) return done(err)
        if(!row) return done(404)

        return done(null, {
            article_id: row.article_id,
            title: row.title,
            author: row.author,
            date_published: new Date(row.date_published).toLocaleDateString(),
            date_edited: new Date(row.date_edited).toLocaleDateString(),
            article_text: row.article_text
        })
    })
```

2. In the controller, get the path parameter and pass it into the model function. Remember to check the error for those 404 digits

```
const getOne = (req, res) => {
    let article_id = parseInt(req.params.article_id);

    articles.getSingleArticle(article_id, (err, result) => {
        if(err === 404) return res.sendStatus(404)
        if(err) return res.sendStatus(500)

        return res.status(200).send(result)
    })
}
```

3. Save and test

Exercise 4.7: Updating an article

1. For the model function, we execute a simple UPDATE statement, passing in the ID to operate on and the article object

```
const updateArticle = (id, article, done) => {
    const sql = 'UPDATE articles SET title=?, author=?, article_text=? WHERE article_id=?'
    let values = [article.title, article.author, article.article_text, id];

    db.run(sql, values, (err) => {
        return done(err)
    })
}
```

2. In the controller, we first want to get the current article details that are stored in the database. For this we can use the getSingleArticle function that we created in the last exercise

```
const updateArticle = (req, res) => {
    let article_id = parseInt(req.params.article_id);

    articles.getSingleArticle(article_id, (err, result) => {
        if(err === 404) return res.sendStatus(404);
        if(err) return res.sendStatus(500);

        //rest of code here...


    })

}
```

3. Next, we want to validate the incoming data using Joi, and return a 400 if the data in the request body is malformed

```
const updateArticle = (req, res) => {
    let article_id = parseInt(req.params.article_id);

    articles.getSingleArticle(article_id, (err, result) => {
        if(err === 404) return res.sendStatus(404);
        if(err) return res.sendStatus(500);

        const schema = Joi.object({
            "title": Joi.string(),
            "author": Joi.string(),
            "article_text": Joi.string()
        })

        const { error } = schema.validate(req.body);
        if(error) return res.status(400).send(error.details[0].message);

        //rest of code here...


    })

}
```

4. We want to find out what specifically is being updated by the user. For this, we can check the request body for each possible property (title, author and article_text) and compare them against what's currently stored in the database (the result that came back from getSingleArticle)

```
        if(error) return res.status(400).send(error.details[0].me

        if(req.body.hasOwnProperty("title")){
            result.title = req.body.title
        }

        if(req.body.hasOwnProperty("author")){
            result.author = req.body.author
        }

        if(req.body.hasOwnProperty("article_text")){
            result.article_text = req.body.article_text
        }

        //rest of code here...


    })
```

5. Finally, we call the model function to update the article with the new data

```
articles.updateArticle(article_id, result, (err, id) => {
    if(err) {
        console.log(err)
        return res.sendStatus(500)
    }

    return res.sendStatus(200)
})
)
```

6. Save and test

Exercise 4.8: Deleting an article

Finally, we have the delete method. Using the code you have already written to help you, implement the delete endpoint.