Hands-on lab: Part 2: Car Inventory Back-end Service

Estimated time needed: 75 minutes

In this lab, you will enhance your application by developing a new back-end car inventory microservice and integrating it with the Django app back end.

Objectives

In this lab, you will:

- Develop a new back-end car inventory microservice using MongoDB and Node.js.
- Integrate this microservice with the Django app back end and start the back-end server successfully.

Prerequisites

1. You must have completed Hands-on lab: Part 1: Front-end Enhancement.
2. Sentiment Analyzer service deployed on Code Engine should be deployed and accessible. Please refer to the section `Deploy sentiment analysis on Code Engine as a microservice` in the lab Create Django Proxy Services Of Backend APIs for the same.
3. Back-end service with Express-MongoDB should be running on one of the terminals. Please refer to the lab Implement API endpoints using Express-Mongo for the same.
4. The front end of the application should have been built. Please refer to the section `Build the client-side and configure it` in the lab User Management for the same.
5. The Django (main application) server should be running. Please refer to the section `Environment setup` in the lab Build

CarModel and CarMake Django Models for the same.

Develop a new back-end car inventory microservice using MongoDB and Node.js

1. Navigate to the project directory

1. Open a new terminal window and navigate to the `xrwvm-fullstack_developer_capstone/server` directory.
2. Generate a new directory named `carsInventory`.

   ```
   mkdir carsInventory
   ```

2. Configure the `package.json` file and install the app dependencies

1. Initialize a new Node.js project and create a `package.json` file inside the `carsInventory` directory by executing the below command:

   ```
   npm init
   ```

2. Add the below app dependencies into package.json:

```
"cors": "^2.8.5",
"express": "^4.18.2",
"mongodb": "^6.3.0",
"mongoose": "^8.0.1"
```

- These dependencies are essential for enabling CORS (Cross-Origin Resource Sharing), handling web application routing and middleware (Express), interacting with MongoDB (MongoDB driver), and providing a convenient way to model data in Node.js applications that use MongoDB (Mongoose).
- The ^ symbol before the version number in each dependency allows for the installation of compatible future updates when running the `npm install` command.
- Set the "name" as `carsInventory`, and the value of "main" as `app.js`.
- With this, your `package.json` file should look similar to this:

```
{
  "name": "carsInventory",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
```

```
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "mongodb": "^6.3.0",
    "mongoose": "^8.0.1"
  }
}
```

3. Install these dependencies by executing the command:

```
npm install
```

3. Set up a MongoDB schema in a file called `inventory.js`

1. You will now set up a MongoDB schema using the `mongoose` library for a collection named `cars`.
2. This will be used to create a 'mongoose' model for interacting with the MongoDB database, allowing the application to perform CRUD operations on car documents in a more structured and organized manner.
3. The schema should define the structure of car documents, which should include fields and their data types as follows:

`dealer_id` : Number

`make`: String

`model`: String

`bodyType`: String

`year`: Number

`mileage`: Number

`price`: Number

4. The model will be named `cars` , corresponding to the 'cars' collection in the connected MongoDB database. The MongoDB model will be exported with this name.

▼ Click to see the solution

```
const { Int32 } = require('mongodb');
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const cars = new Schema({
dealer_id: {
    type: Number,
    required: true
},
make: {
    type: String,
    required: true
  },
model: {
    type: String,
    required: true
  },
bodyType: {
    type: String,
    required: true
  },
year: {
    type: Number,
    required: true
  },
mileage: {
    type: Number,
    required: true
```

```
  },
price: {
    type: Number,
    required: true
  }
});
module.exports = mongoose.model('cars', cars);
```

4. Obtain the JSON data set with the car inventory and associated details

1. Start by creating a folder named `data` and navigating into it:

```
mkdir data
cd data
```

2. Download the car inventory data set:

```
wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-CD0321EN-SkillsNetwork/labs/v2/m6/car_records.json
```

3. Inspect the file and observe that it presents the following fields for each car object:

```
make, model, bodyType, year, dealer_id, mileage, price
```

Switch back to the `carsInventory` directory to create further files.

5. Establish the Node.JS server featuring the back-end endpoints

1. Start by creating a file named `app.js`:

```
touch app.js
```

2. **It should perform the following functionality:**

- Set up an Express server with MongoDB integration.
- Retrieve data from the file `car_records.json`.
- Establish a connection to MongoDB
- Defines the root API endpoint `/` that responds with a `Welcome to the Mongoose API` message when the root of the API is accessed.
- Defines the below six endpoints for querying cars based on various criteria
  i. A `cars/:id` endpoint that retrieves and returns car documents from the MongoDB collection based on the specified dealer ID.
  ii. A `/carsbymake/:id/:make`endpoint that retrieves and returns car documents based on both dealer ID and car make.
  iii. A `/carsbymake/:id/:model` endpoint that retrieves and returns car documents based on both dealer ID and car model.
  iv. A `/carsbymaxmileage/:id/:mileage`endpoint that retrieves and returns car documents based on both dealer ID and mileage constraints as below:

Mileage:

- Less than or equal to 50000
- 50000 to 100000
- 100000 to 150000
- 150000 to 200000
- Greater than 200000

v. A `/carsbyprice/:id/:price` endpoint that retrieves and returns car documents based on both dealer ID and price constraints as below:

Price:

- Less than or equal to 20000
- 20000 to 40000
- 40000 to 60000
- 60000 to 80000
- Greater than 80000

vi. A `/carsbymake/:id/:year` endpoint that retrieves and returns car documents based on both dealer ID and a minimum year

constraint.

- Start the server on port 3050.

▼ Click to see the solution

```
/*jshint esversion: 8 */
const express = require('express');
const mongoose = require('mongoose');
const fs = require('fs');
const cors = require('cors');
const app = express();
const port = 3050;
app.use(cors());
app.use(express.urlencoded({ extended: false }));
const carsData = JSON.parse(fs.readFileSync('car_records.json', 'utf8'));
mongoose.connect('mongodb://mongo_db:27017/', { dbName: 'dealershipsDB' })
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.error('MongoDB connection error:', err));
const Cars = require('./inventory');
try {
  Cars.deleteMany({}).then(() => {
    Cars.insertMany(carsData.cars);
  });
} catch (error) {
  console.error(error);
  // Handle errors properly here
}
app.get('/', async (req, res) => {
  res.send('Welcome to the Mongoose API');
});
app.get('/cars/:id', async (req, res) => {
  try {
    const documents = await Cars.find({dealer_id: req.params.id});
    res.json(documents);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching reviews' });
  }
});
app.get('/carsbymake/:id/:make', async (req, res) => {
  try {
    const documents = await Cars.find({dealer_id: req.params.id, make: req.params.make});
    res.json(documents);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching reviews by car make and model' });
  }
```

```
});
app.get('/carsbymodel/:id/:model', async (req, res) => {
  try {
    const documents = await Cars.find({ dealer_id: req.params.id, model: req.params.model });
    res.json(documents);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching dealers by ID' });
  }
});
app.get('/carsbymaxmileage/:id/:mileage', async (req, res) => {
  try {
    let mileage = parseInt(req.params.mileage)
    let condition = {}
    if(mileage === 50000) {
      condition = { $lte : mileage}
    } else if (mileage === 100000){
      condition = { $lte : mileage, $gt : 50000}
    } else if (mileage === 150000){
      condition = { $lte : mileage, $gt : 100000}
    } else if (mileage === 200000){
      condition = { $lte : mileage, $gt : 150000}
    } else {
      condition = { $gt : 200000}
    }
    const documents = await Cars.find({ dealer_id: req.params.id, mileage : condition });
    res.json(documents);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching dealers by ID' });
  }
});
app.get('/carsbyprice/:id/:price', async (req, res) => {
    try {
        let price = parseInt(req.params.price)
        let condition = {}
        if(price === 20000) {
          condition = { $lte : price}
        } else if (price=== 40000){
          console.log("\n \n \n "+ price)
          condition = { $lte : price, $gt : 20000}
        } else if (price === 60000){
          condition = { $lte : price, $gt : 40000}
        } else if (price === 80000){
          condition = { $lte : price, $gt : 60000}
        } else {
          condition = { $gt : 80000}
        }
        const documents = await Cars.find({ dealer_id: req.params.id, price : condition });
```

```
        res.json(documents);
      } catch (error) {
        res.status(500).json({ error: 'Error fetching dealers by ID' });
      }
});
app.get('/carsbyyear/:id/:year', async (req, res) => {
  try {
    const documents = await Cars.find({ dealer_id: req.params.id, year : { $gte :req.params.year }});
    res.json(documents);
  } catch (error) {
    res.status(500).json({ error: 'Error fetching dealers by ID' });
  }
});
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

6. Create a Docker image and start the microservices server

Create a Docker image for the Node.js application

1. Create a `Dockerfile`

   ```
   touch Dockerfile
   ```

2. The following will be its contents:

- Base Docker image.: `node:18.12.1-bullseye-slim`
- Install npm version `9.1.3`.
- Add `package.json`, `app.js`, `car_records.json` from the current working directory to the root directory of the Docker image.
- Copy all files from the current working directory to the current directory of the Docker image.
- Expose the container to listen on port 3050.
- Specify the default command to run when the container starts as: `node app.js`.

▼ Click to see the solution

```
FROM node:18.12.1-bullseye-slim
RUN npm install -g npm@9.1.3
ADD package.json .
ADD app.js .
ADD data/car_records.json .
COPY . .
RUN npm install
EXPOSE 3050
CMD [ "node", "app.js" ]
```

Setup a Docker Compose configuration file for running two services (MongoDB container and Node.js application)

1. Create a Docker Compose configuration YAML file (`docker-compose.yml`).

▼ Click to see the command

```
touch docker-compose.yml
```

2. Paste the following content into it:

```
version: '3.9'
services:
  # Mongodb service
  mongo_db:
    container_name: carsInventory_container
    image: mongo:latest
    ports:
      - 27018:27017
    restart: always
    volumes:
      - mongo_data:/data/db
  # Node api service
  api:
    image: nodeapp
    ports:
      - 3050:3050
    depends_on:
      - mongo_db
volumes:
  mongo_data: {}
```

▼ Click to see the functionality of the code

```
version: '3.9'
```

- Specifies the version of the Docker Compose file syntax.

mongo_db (MongoDB Service):

```
container_name: carsInventory_container:
```

- Sets the name of the MongoDB container to carsInventory_container.

```
image: mongo:latest
```

- Specifies the use of the latest version of the official MongoDB Docker image.

```
ports: - 27018:27017
```

- Maps port 27018 on the host to port 27017 on the MongoDB container, allowing external access to the MongoDB service.

```
restart: always
```

- Configures the container to restart automatically if it stops unexpectedly.

```
volumes: - mongo_data:/data/db
```

- Creates a Docker volume named mongo_data to persistently store MongoDB data.

api (Node API Service):

```
image: nodeapp
```

- Specifies the Docker image for the Node.js application (assumed to be named ‘nodeapp’).

```
ports: - 3050:3050
```

- Maps port 3050 on the host to port 3050 on the Node.js application container.

```
depends_on: - mongo_db
```

- Ensures that the Node.js application service starts only after the MongoDB service (mongo_db) is up and running.

Volumes Section:

```
volumes: mongo_data: {}
```

- Defines the Docker volume named mongo_data. This volume is used by the MongoDB service to persistently store data in the /data/db directory.

3. Build the Docker application.

▼ Click to see the command

```
docker build . -t nodeapp
```

4. Execute the below command to launch the server:

▼ Click to see the command

```
docker-compose up
```

5. Launch the server and verify if the root endpoint of the Cars inventory service presents the message `Welcome to the Mongoose API.`

   Remember to save this endpoint, as you'll integrate it with the Django server back end in the next segment.

6. Test the output of the following endpoints:

```
cars/:id
/carsbymake/:id/:make
/carsbymodel/:id/:model
/carsbymaxmileage/:id/:mileage
/carsbyprice/:id/:price
/carsbyyear/:id/:year
```

Ensure the server keeps running, as it's crucial for testing the output of the Django app in the next section.

Integrate the microservice with the Django app back-end and startup the back-end server successfully

1. Navigate to the `xrwvm-fullstack_developer_capstone/server`directory.

Stop the Django server, if it is already running.

2. Insert the URL of the cars inventory service endpoint (copied in the previous section) into the `.env` file.

   ```
   searchcars_url='your end'
   ```

Note: Exclude quotes `" "` around the URL, and be careful not to copy the `\` at the end.

3. Include the code for performing the following functionality into `restapis.py`.

A. Fetch the URL from the .env file.

```
searchcars_url = os.getenv(
    'searchcars_url',
    default="http://localhost:3050/")
```

B. Implement a method named `searchcars_request` with the following functionalities:

- Accept an endpoint and variable keyword arguments.
- Construct a complete request URL utilizing the provided endpoint, query parameters, and a base URL.
- Execute a GET request and return the JSON content of the response.
- Handle errors, including network exceptions, and provide a success completion message.

The code structure will closely resemble the `get_request` method you previously created in the main Capstone project.

▾ Click to see the solution

```
def searchcars_request(endpoint, **kwargs):
    params = ""
    if (kwargs):
```

```
    for key, value in kwargs.items():
        params = params+key + "=" + value + "&"
request_url = searchcars_url+endpoint+"?"+params
print("GET from {} ".format(request_url))
try:
    # Call get method of requests library with URL and parameters
    response = requests.get(request_url)
    return response.json()
except:
    # If any error occurs
    print("Network exception occurred")
finally:
    print("GET request call complete!")
```

4. Add a view named `get_inventory`in the `djangoapp/views.py` file to fetch the car inventory.

- This view should process an HTTP request, extracting query parameters and a dealer ID.
- It should construct an API endpoint using the provided parameters and invoke the `searchcars_request` function, defined in restapis.py, to retrieve car data. Make sure to incorporate the module import for the `searchcars_request` function.
- It should return a JSON response with a status of 200 and the fetched cars if the dealer ID is provided.
- If no dealer ID is present, it should return a JSON response with a status of 400 and a `Bad Request` message.

▼ Click to see the solution

```
# Module import
from .restapis import get_request, analyze_review_sentiments, post_review, searchcars_request
# Code for the view
def get_inventory(request, dealer_id):
    data = request.GET
    if (dealer_id):
        if 'year' in data:
            endpoint = "/carsbyyear/"+str(dealer_id)+"/"+data['year']
        elif 'make' in data:
            endpoint = "/carsbymake/"+str(dealer_id)+"/"+data['make']
```

```
        elif 'model' in data:
            endpoint = "/carsbymodel/"+str(dealer_id)+"/"+data['model']
        elif 'mileage' in data:
            endpoint = "/carsbymaxmileage/"+str(dealer_id)+"/"+data['mileage']
        elif 'price' in data:
            endpoint = "/carsbyprice/"+str(dealer_id)+"/"+data['price']
        else:
            endpoint = "/cars/"+str(dealer_id)

        cars = searchcars_request(endpoint)
        return JsonResponse({"status": 200, "cars": cars})
    else:
        return JsonResponse({"status": 400, "message": "Bad Request"})
    return JsonResponse({"status": 400, "message": "Bad Request"})
```

5. Include the route for this view in the `djangoapp/urls.py` file.

▼ Click to see the solution

```
path(route='get_inventory/<int:dealer_id>', view=views.get_inventory, name='get_inventory'),
```

6. Navigate to the `xrwvm-fullstack_developer_capstone/server`directory.

Make sure the Docker Compose server is running. If not, start it using the command `docker-compose up`.

7. Perform model migrations and start the server.

▼ Click to see the commands

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

8. Ensure the server starts successfully without errors. If any error logs appear, review and address your code as needed.

Since you have only constructed the back end of the Cars inventory microservice, testing the output on the Django app is not applicable at this stage.
Following the construction of the front end in the upcoming lab (Part 3 of enhancing your application), you will be able to assess the Django app output.

Conclusion

Congratulations on completing this lab!

In this lab, you established a new back-end microservice for obtaining various details pertaining to car inventory through the utilization of MongoDB and Node.js servers.
You then integrated the newly created microservice with the Django app back end and verified the successful startup of the back-end server.
These steps enhanced the back-end components of your application.

Author(s)

Lavanya TS
K Sundararajan