Practical: REST API with Multi-Container Service Architecture (JavaScript)

In this practical, you will be hands on with a basic REST API implemented with Node, JavaScript, Express and MariaDB.

Your task is to review the code, containerise the application and deploy it on EC2.

Importantly, this application uses two containers: one for the Node process, another for the MariaDB database.

The application is a basic task manager. It does not implement any CPU-intensive task but a starting point for a REST API and a service architecture having multiple containers.

Table of Contents

Step 1: Understand the context

This is a task management app that has a REST API with five endpoints:

1. Get all tasks
2. Get a particular task
3. Create a task
4. Update a task
5. Delete a task

There is no separation of tasks by user, and no security, so anyone could be interacting with this API and be editing/deleting other peoples tasks. Obviously this does not make sense in the real-world but this prac is not about demonstrating separation of users.

We have provided a simple API to get started, just as a demonstration of how an Express-based API can be implemented and how to orchestrate "one application" that has multiple applications.

Database access with JavaScript

Database methods to create, find, update, or delete records are asynchronous. This means that the database querying methods return immediately, and the code to handle the success or failure of them run at a later time when the query completes. Other code can execute while the server is waiting for the database operation to complete, so the server can remain responsive to other requests.

JavaScript has a number of mechanisms for supporting asynchronous behaviour. Historically JavaScript relied heavily on passing callback functions to asynchronous methods to handle the success and error cases. In modern JavaScript callbacks have largely been replaced by Promises. Promises are objects that are (immediately) returned by an asynchronous method that represent its future state. When the operation completes, the promise object is "settled", and resolves an object that represents the result of the operation or an error.

Step 2: Implement the Task Manager API

You should follow along with our instructions. We have not provided a ZIP of the code because you need to get hands-on with setting-up a project like this one. It is relatively small, to keep the efforts low. The important part is you understanding how the app is structured and what the code does. Remember, CAB432 is about cloud infrastructure, not software development (i.e., learning a programming language or building small apps). We need to focus on *how to design and deploy apps that are scalable*.

We are going to get started by initialising a new Node project.

```
npm init -y
npm install express mariadb
```

Create a layered architecture using.

- routes
- controllers
- models

> Read more about routers and controllers here: [MDN Web Docs: Routes and Controllers](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

```
mkdir -p src/{routes,controllers,models,middleware}
touch src/routes/tasks.js
touch src/controllers/tasks.js
touch src/models/task.js
touch src/db.js
touch app.js
```

The main entry-point of the app is `app.js`:

```
const express = require('express');
const app = express();

const tasksRouter = require('./src/routes/tasks');

app.use(express.json());
app.use('/tasks', tasksRouter);

const PORT = 3000;
app.listen(PORT, () => console.log(`Server running on http://localhos
t:${PORT}`));
```

Here is `src/db.js` which establishes the connection to the database.

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

const pool = mariadb.createPool({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'user',
  password: process.env.DB_PASSWORD || 'pass',
  database: process.env.DB_NAME || 'tasksdb',
  connectionLimit: 5,
});

// Init logic without messing with exports
(async () => {
  let conn;
  try {
    conn = await pool.getConnection();
    console.log('Connection successful.');
    await conn.query(`
      CREATE TABLE IF NOT EXISTS tasks (
        id INT AUTO_INCREMENT PRIMARY KEY,
        title VARCHAR(255) NOT NULL,
        completed BOOLEAN DEFAULT FALSE
      );
    `);
  } catch (err) {
    console.error('DB init failed:', err.message);
  } finally {
    if (conn) {
      conn.release();
      console.log('Releasing connection...');
    }
  }
})();

module.exports = pool;
```

The next file we will implement is `src/models/task.js`, which is a model to represent tasks. It is responsible for the main interactions with the database.

```
const pool = require('../db');

exports.getAll = async () => {
  const conn = await pool.getConnection();
  const rows = await conn.query('SELECT * FROM tasks');
  conn.release();
  return rows;
};

exports.getById = async (id) => {
  const conn = await pool.getConnection();
  const rows = await conn.query('SELECT * FROM tasks WHERE id = ?',
[id]);
  conn.release();
  return rows[0];
};

exports.create = async (title) => {
  const conn = await pool.getConnection();
  const result = await conn.query('INSERT INTO tasks (title, complete
d) VALUES (?, ?)', [title, 0]);
  conn.release();
  return { id: Number(result.insertId), title, completed: 0 };
};

exports.update = async (id, title, completed) => {
  const conn = await pool.getConnection();
  const result = await conn.query(
    'UPDATE tasks SET title = ?, completed = ? WHERE id = ?',
    [title, completed ? 1 : 0, id]
  );
  conn.release();
  return { updated: result.affectedRows > 0 };
};

exports.remove = async (id) => {
  const conn = await pool.getConnection();
  const result = await conn.query('DELETE FROM tasks WHERE id = ?',
[id]);
  conn.release();
  return { deleted: result.affectedRows > 0 };
};
```

And a controller, `src/controllers/tasks.js` that implements the logic around the model. Notice that this controller handles the incoming request (`req`) and outgoing response (`res`), and has no direct interaction with the database -- the data model does that. This layered approach is useful for if/when the model is changed (we wouldn't need to update any controllers that interact with the Task model) or when the underlying database engine changes (we only update the models' interaction with the database, not any controllers). The controller also performs some input validation because it is capable of returning error messages to the requester.

```
const Task = require('../models/task');

exports.getAllTasks = (req, res) => {
  Task.getAll()
    .then(rows => res.json(rows))
    .catch(err => res.status(500).json({ error: err.message }));
};

exports.getTaskById = (req, res) => {
  Task.getById(req.params.id)
    .then(row => {
      if (!row) return res.status(404).json({ error: 'Task not found'
});
      res.json(row);
    })
    .catch(err => res.status(500).json({ error: err.message }));
};

exports.createTask = (req, res) => {
  const { title } = req.body;
  if (!title) return res.status(400).json({ error: 'Title is require
d' });

  Task.create(title)
    .then(task => res.status(201).json(task))
    .catch(err => res.status(500).json({ error: err.message }));
};

exports.updateTask = (req, res) => {
  const { title, completed } = req.body;

  Task.update(req.params.id, title, completed)
    .then(result => {
      if (!result.updated) return res.status(404).json({ error: 'Task
not found' });
      res.json({ message: 'Task updated' });
    })
    .catch(err => res.status(500).json({ error: err.message }));
};

exports.deleteTask = (req, res) => {
  Task.remove(req.params.id)
    .then(result => {
      if (!result.deleted) return res.status(404).json({ error: 'Task
not found' });
      res.json({ message: 'Task deleted' });
    })
    .catch(err => res.status(500).json({ error: err.message }));
};
```

Now that we have a model and controller, we need to connect these to the interface of the Task Manager app. Our UI is going to be a REST HTTP API, therefore, we need to tell the Express app what are our endpoints/routes (i.e., HTTP URLs) for reaching the controllers.

For instance, we may send a request to update Task #123 with the following HTTP request: `PUT https://cab432.com/tasks/123`. Its payload (i.e., HTTP body) may be something like `{ title: "Task Renamed", completed: 1 }`. Here is `src/routes/tasks.js`:

```
const express = require('express');
const router = express.Router();
const controller = require('../controllers/tasks');

router.get('/', controller.getAllTasks);
router.get('/:id', controller.getTaskById);
router.post('/', controller.createTask);
router.put('/:id', controller.updateTask);
router.delete('/:id', controller.deleteTask);

module.exports = router;
```

We have now provided code for all the necessary files. However, you won't yet be able to run this app because there is no database available. The next step is to create relevant Docker containers: one for the Node app and one that is running MariaDB.

Step 3: Deploy with Docker

This section is about setting up the app to run with Docker. At the bottom of this prac, in the appendix, are some extra Docker commands that you might find useful along the way (in case things go wrong or to help with debugging). The appendix also includes a command to launch a temporary container that is running a web-based database explorer, so you can manually create/update/delete records in the MariaDB.

Let's start by creating a container for the Node app.

Create a Dockerfile at the root of the project (i.e., create `Dockerfile` in the same directory as `app.js`):

```
FROM node:20-alpine

WORKDIR /usr/src/app

COPY package*.json ./
RUN npm install

COPY . .

CMD ["node", "app.js"]
```

Build that container

```
docker build -t task-api .
```

Create a network (read docs **here** (https://docs.docker.com/engine/network/)) so the containers can communicate

```
docker network create tasknet
```

Run MariaDB as a container

```
docker run -d --name mariadb --network tasknet -e MARIADB_ROOT_PASSWO
RD=rootpass -e MARIADB_DATABASE=tasksdb -e MARIADB_USER=user -e MARIA
DB_PASSWORD=pass -p 3306:3306 -v mariadb-data:/var/lib/mysql mariadb:
11
```

- `docker run -d`: Runs the container in **detached mode** (in the background).
- `--name mariadb`: Assigns the container the name `mariadb`.
- `--network tasknet`: Connects the container to a custom Docker network named `tasknet`.
- **Environment variables (`-e`):**
  - `MARIADB_ROOT_PASSWORD=rootpass`: Sets the root password for MariaDB.
  - `MARIADB_DATABASE=tasksdb`: Creates a new database named `tasksdb`.
  - `MARIADB_USER=user`: Creates a new user named `user`.
  - `MARIADB_PASSWORD=pass`: Sets the password for the new user.
- `-p 3306:3306`: Maps port **3306** on the host to **3306** in the container (MariaDB's default port).
- `-v mariadb-data:/var/lib/mysql`: Mounts a **named volume** (`mariadb-data`) to persist database data.
- `mariadb:11`: Specifies the **MariaDB image version 11** to use.

Run the Task Manager container

```
docker run -d --name task-api --network tasknet -p 3000:3000 -e DB_HO
ST=mariadb -e DB_USER=user -e DB_PASSWORD=pass -e DB_NAME=tasksdb tas
k-api
```

- `docker run -d`: Runs the container in **detached mode** (background).
- `--name task-api`: Names the container `task-api`.
- `--network tasknet`: Connects the container to the `tasknet` Docker network.
- `-p 3000:3000`: Maps port **3000** on the host to **3000** in the container (used by the API).
- **Environment variables (`-e`):**
  - `DB_HOST=mariadb`: Specifies the database host (the MariaDB container).
  - `DB_USER=user`: Sets the database username.
  - `DB_PASSWORD=pass`: Sets the database password.
  - `DB_NAME=tasksdb`: Specifies the database name to connect to.
- `task-api`: Uses the `task-api` image to run the container.

The two containers should now be running. Check that by running

```
docker container ls -a
```

Step 4: Test the API

Try all endpoints

```
curl http://localhost:3000/tasks
```

Appendix

```
# View running containers
docker ps

# View logs for MariaDB (or any container)
docker logs mariadb

# Inspect container details (network, volumes, etc.)
docker inspect mariadb

# View container resource usage
docker stats

# Stop a container
docker stop mariadb

# Remove a container
docker rm mariadb

# Remove the Docker network
docker network rm tasknet

# List all Docker networks
docker network ls

# Run an interactive container on the same network for debugging
docker run -it --rm --network tasknet alpine sh

# Run a temporary container with Adminer, a web-based database explor
er.
# Access it at http://localhost:8080
# Connect using:
#   System:    MariaDB
#   Server:    mariadb
#   Username:  user
#   Password:  pass
#   Database:  tasksdb
docker run --rm -d --name adminer --network tasknet -p 8080:8080 admi
ner
```