

REST APIS

CSE5006 – LAB 7

REPOSITORY:

[HTTPS://GITHUB.COM/CSE5006/LAB-7](https://github.com/CSE5006/LAB-7)

TABLE OF CONTENTS

1. What is REST?	4
2. Making Requests with HTTPie	4
2.1. Git Clone	4
2.2. Get all Posts	5
2.3. Update an Author	7
2.4. Create a New Post.....	7
2.5. Delete a Post	8
3. Creating a REST API.....	10
3.1. Fork Blog Repository	10
3.2. Clone Blog Repository.....	11
3.3. Boilerplate	11
3.4. Start the Container	11
3.5. Web Browser	11
3.6. Browse Project Files.....	12
4. Routes and Controllers	13
4.1. Controller Routing.....	15
4.1.1. 404 Not Found	15
4.1.2. Connect Posts Controller	15
4.1.3. Try Again	16
4.2. HTTP Methods	16
4.2.1. Return	16
4.2.2. Pay Attention.....	16
4.2.3. Two Requests	17
4.2.4. Exercise 1	18
5. Creating and Linking a Database Container	19
5.1. Environment Variable Configuration.....	19
5.1.1. New Folder.....	19
5.1.2. New File	20
5.1.3. Random Password	20
5.1.4. Edit and Past	20
5.2. Docker Compose Service	20
5.2.1. Stop.....	20
5.2.2. New Service	21
5.2.3. Try It	22
6. Using the Database in Node	22

6.1. Configuring the Database Connection	22
6.1.1. New File	24
6.1.2. Another New File	24
6.1.3. Create Subdirectories	26
6.2. Creating a Model	26
6.2.1. Run the Command	26
6.2.2. Modify Title and Content	28
6.2.3. Add Table	29
6.2.4. Definition	30
6.3. Creating a Seeder	30
6.3.1. Create Seeder	30
6.3.2. Up Function	32
6.3.3. Seed Dummy Records	34
6.4. Using the Model from the Controller	34
6.4.1. Create Contents	36
6.4.2. Post Model	36
6.4.3. Commit	39
6.4.4. Exercise 2	39
6.4.5. Exercise 3	39

1. WHAT IS REST?

Representational state transfer (REST) is an extremely popular way of architecting web applications which are used by other pieces of software. An important property of REST is that it's "stateless". This means that there is no per-client "context" stored on the server which affects how the server responds to requests - it is the client's responsibility to send all information required by the server along with the request. In simple terms, the server does not "know about" the client making the request. This property makes it much easier to scale RESTful applications, since requests needn't be handled by the same machine for fear of losing some important state information stored in memory.

REST makes extensive use of different HTTP methods for different tasks. The "default" HTTP method is GET, which is sent to the server when you browse to a website. That is, when you enter `http://www.google.com/` in your browser, the server for `www.google.com` is sent a GET request for the root index path, `/`. In REST, multiple HTTP methods are used for different purposes.

<i>HTTP Method</i>	<i>Purpose</i>
<i>GET</i>	Show or list resources
<i>POST</i>	Create a resource
<i>PUT</i>	Update a resource
<i>DELETE</i>	Delete a resource

For example, creating a new tweet would be a POST. Editing a blog post would be a PUT. Loading a LinkedIn profile would be a GET. Deleting your Facebook profile so you can move to the alps and train as a monk would be a DELETE.

2. MAKING REQUESTS WITH HTTPIE

We will now look at what using a REST API is like from the client's perspective. To do this we will be using a command-line tool called HTTPie and a simple fake REST server.

2.1. GIT CLONE

Use Git to clone the lab 7 repository from <https://github.com/CSE5006/lab-7> (there is no need to make your own fork). Bring this server up using Docker compose while inside the **lab-7/mock-rest-server/** directory (refer to the previous lab if you forgot how to do this). Now we can start playing with the API!

```
vboxuser@CSE5006:~/Documents$ git clone https://github.com/CSE5006/lab-7
Cloning into 'lab-7'...
remote: Enumerating objects: 31, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 31 (delta 6), reused 30 (delta 5), pack-reused 0
Receiving objects: 100% (31/31), 4.68 KiB | 1.17 MiB/s, done.
Resolving deltas: 100% (6/6), done.
vboxuser@CSE5006:~/Documents$ cd lab-7
vboxuser@CSE5006:~/Documents/lab-7$
```

```
vboxuser@CSE5006:~/Documents/lab-7$ cd mock-rest-server/
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ docker compose up --build
[+] Building 3.1s (5/11)
=> [api internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 902B 0.0s
=> [api internal] load .dockerignore 0.2s
=> => transferring context: 2B 0.0s
=> [api internal] load metadata for docker.io/library/node:18.16.0-alpin 2.0s
=> CACHED [api 1/7] FROM docker.io/library/node:18.16.0-alpine@sha256:90 0.0s
=> [api internal] load build context 0.4s
=> => transferring context: 1.20kB 0.0s
=> [api 2/7] RUN apk add --no-cache tini curl bash sudo 0.8s
```

```
mock-rest-server-api-1 |
mock-rest-server-api-1 |
mock-rest-server-api-1 | \{^_^\}/ hi!
mock-rest-server-api-1 |
mock-rest-server-api-1 | Loading db.json
mock-rest-server-api-1 |
mock-rest-server-api-1 | Done
mock-rest-server-api-1 |
mock-rest-server-api-1 | Resources
mock-rest-server-api-1 | http://localhost:3000/posts
mock-rest-server-api-1 |
mock-rest-server-api-1 | Home
mock-rest-server-api-1 | http://localhost:3000
mock-rest-server-api-1 |
mock-rest-server-api-1 | Type s + enter at any time to create a snapshot of t
he database
mock-rest-server-api-1 | Watching...
mock-rest-server-api-1 |
mock-rest-server-api-1 |
```

2.2. GET ALL POSTS

Make a GET request to the **/posts** endpoint:

```
http GET localhost:3000/posts
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http GET localhost:3000/posts
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 143
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:24:58 GMT
ETag: W/"8f-SbWRw/KVsfpvEjuTAjPnM0W2GjE"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

[
  {
    "author": "John",
    "id": 1,
    "title": "First post"
  },
  {
    "author": "Fred",
    "id": 2,
    "title": "Second post"
  }
]
```

This should present you with a list of posts in JSON (JavaScript Object Notation) format. JSON is a format which represents data (usually an array or an object) in JavaScript syntax.

2.3. UPDATE AN AUTHOR

Let's change the author of the first post to your name. So, if your name is Jim:

```
http PUT localhost:3000/posts/1 title="First post" author="Jim"
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http PUT localhost:3000/posts/1 title="First post" author="Jim"
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 57
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:32:45 GMT
ETag: W/"39-xbv01Wfi1Jerd2RL4CUMS5JSq68"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

{
  "author": "Jim",
  "id": 1,
  "title": "First post"
}
```

Now, when you retrieve the first post, the data response should reflect the change you made.

```
http GET localhost:3000/posts/1
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http GET localhost:3000/posts/1
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 57
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:33:33 GMT
ETag: W/"39-xbv01Wfi1Jerd2RL4CUMS5JSq68"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

{
  "author": "Jim",
  "id": 1,
  "title": "First post"
}
```

2.4. CREATE A NEW POST

```
http POST localhost:3000/posts title="Rings" author="Sauron"
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http POST localhost:3000/posts title="Rings" author="Sauron"
HTTP/1.1 201 Created
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 55
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:34:09 GMT
ETag: W/"37-ScYsWRG80lFXrKLxL/siXRbXtkQ"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, X-HTTP-Method-Override, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

{
  "author": "Sauron",
  "id": 3,
  "title": "Rings"
}
```

Notice how when we want to create a new resource, we don't need to specify the ID. The server will automatically allocate an available ID to the resource for us. List the posts again to see this for yourself:

```
http GET localhost:3000/posts
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http GET localhost:3000/posts
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 209
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:35:32 GMT
ETag: W/"d1-0RM0NpjoOsFAGZ+4B1vEx6mLDB4"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

[
  {
    "author": "Jim",
    "id": 1,
    "title": "First post"
  },
  {
    "author": "Fred",
    "id": 2,
    "title": "Second post"
  },
  {
    "author": "Sauron",
    "id": 3,
    "title": "Rings"
  }
]
```

2.5. DELETE A POST

Finally, we can wipe Sauron from the face of Middle Earth (well, from our application at least) by using DELETE. Here's the command to do so, assuming that the post's automatically assigned ID was 3:

```
http DELETE localhost:3000/posts/3
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http DELETE localhost:3000/posts/3
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 2
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:36:05 GMT
ETag: W/"2-vyGp6PvFo4RvsFtPoIWeCReyIC8"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

{}

```

```
http GET localhost:3000/posts
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ http GET localhost:3000/posts
HTTP/1.1 200 OK
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Connection: keep-alive
Content-Length: 142
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:37:18 GMT
ETag: W/"8e-01FjQLWoeNt758M/dNZDqehtfu0"
Expires: -1
Keep-Alive: timeout=5
Pragma: no-cache
Vary: Origin, Accept-Encoding
X-Content-Type-Options: nosniff
X-Powered-By: Express

[
  {
    "author": "Jin",
    "id": 1,
    "title": "First post"
  },
  {
    "author": "Fred",
    "id": 2,
    "title": "Second post"
  }
]

```

Once again list the posts to confirm that the deletion was successful.

As you can see, working with a REST API is quite easy, which is one of its big strengths. Major websites like Facebook and Twitter have similar APIs which make integrating such services into mobile apps very easy.

If you open the other terminal, you can see the API activities when receiving the request. If you did the tasks correctly, you will find the list of activities similar to this output

```
mock-rest-server-api-1 | GET /posts 200 34.680 ms - 143
mock-rest-server-api-1 | GET /posts 200 5.634 ms - 143
mock-rest-server-api-1 | PUT /posts/1 200 19.361 ms - 57
mock-rest-server-api-1 | GET /posts/1 200 5.309 ms - 57
mock-rest-server-api-1 | POST /posts 201 3.760 ms - 55
mock-rest-server-api-1 | GET /posts 200 18.716 ms - 209
mock-rest-server-api-1 | DELETE /posts/3 200 8.066 ms - 2
mock-rest-server-api-1 | GET /posts 200 2.088 ms - 142
```

You may shut down the server when finished by opening another terminal to run this command.

```
docker compose down
```

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ docker compose down
[+] Running 2/2
 ✓ Container mock-rest-server-api-1 Removed
 ✓ Network mock-rest-server_default Removed
```

3. CREATING A REST API

For the rest of this lab we will work on creating a REST API ourselves using Node.js. This will be our first step towards building a fully functioning blog, which will be continued in future labs. Once again you will be expected to make Git commits where appropriate as you progress.

3.1. FORK THE REPOSITORY

To begin you will now need to create a private fork of the lab 7 repository.

3.2. CLONE BLOG REPOSITORY

Clone your newly created fork of the lab 7 repository. You may need to name it something new so the directories don't overlap.

If you cloned the whole repository, the blog folder is already located in the repository.

```
vboxuser@CSE5006:~/Documents/lab-7/mock-rest-server$ cd ..
vboxuser@CSE5006:~/Documents/lab-7$ cd blog/
vboxuser@CSE5006:~/Documents/lab-7/blog$ ls -l
total 8
drwxrwxr-x 3 vboxuser vboxuser 4096 Aug 23 19:21 api
-rw-rw-r-- 1 vboxuser vboxuser 159 Aug 23 19:21 docker-compose.yml
vboxuser@CSE5006:~/Documents/lab-7/blog$
```

3.3. BOILERPLATE

The **blog/api/** folder contains the beginnings of our REST API. Navigate into that directory and open a terminal window and first run the following command:

```
chmod u+x start.sh
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ cd api
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ chmod u+x start.sh
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

3.4. START THE CONTAINER

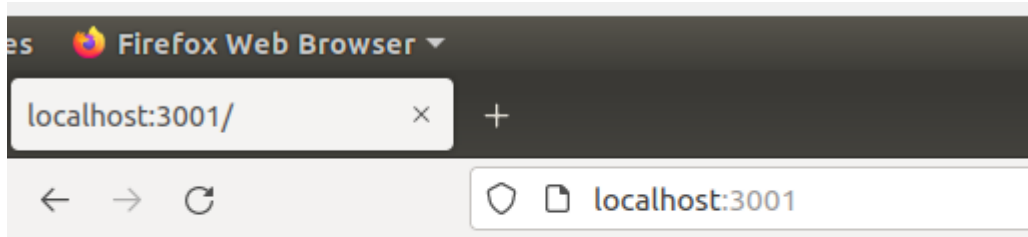
Now start the server by running the usual Docker Compose command. Go to the parent directory before running docker. We do not need to run Gulp since there is no frontend code to transpile (yet).

```
docker compose up --build
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ cd ..
vboxuser@CSE5006:~/Documents/lab-7/blog$ docker compose up --build
[+] Building 2.4s (5/13)
=> [api internal] load .dockerignore                                0.1s
=> => transferring context: 2B                                       0.0s
=> [api internal] load build definition from Dockerfile             0.2s
=> => transferring dockerfile: 1.33kB                                0.0s
=> [api internal] load metadata for docker.io/library/node:18.16.0-alpine 2.0s
=> [api 1/9] FROM docker.io/library/node:18.16.0-alpine@sha256:9036ddb8252ba7089c2c83eb2b0d 0.0s
=> [api internal] load build context                                0.1s
=> CACHED [api 2/9] RUN apk add --no-cache tini curl bash sudo    0.0s
=> [api 3/9] RUN mkdir -p /app                                      0.2s
```

3.5. WEB BROWSER

Point your web browser to **localhost:3001**. You should be presented with a short message.



This is an API. Please hit one of my endpoints, like ``/posts``.

3.6. BROWSE PROJECT FILES

Open the project up in Visual Studio Code and browse the source files to familiarise yourself with the code base. Read the comments and try to gain a little intuition about how some of the pieces fit together.

```
File Edit Selection View Go Run Terminal Help

EXPLORER
LAB07-BLOG
  api
    src
      config
        JS routes.js
      controllers
        JS index.js
        JS posts.js
    JS app.js
    JS server.js
    Dockerfile
    package.json
    README.md
    start.sh
    .gitignore
    docker-compose.yml

JS posts.js
api > src > controllers > JS posts.js > ...
1  const express = require('express');
2
3  const router = express.Router();
4
5  // Index: GET /posts/
6  router.get('/', (req, res) => {
7    res.json({ todo: 'List posts' });
8  });
9
10 // Show: GET /posts/:postId/
11 //
12 // Note that the path contains a variable (
13 // made available as a property of the req
14 router.get('/:postId', (req, res) => {
15   res.json({ todo: 'Show post with ID=' +
16 });
17
18 // Destroy: DELETE /posts/:postId/
19 //
20 // Note that the path is the same as the "S
21 // is different (we are using router.delete
22 router.delete('/:postId', (req, res) => {
23   res.json({ todo: 'Delete post with ID=' +
24 });
25
26 // Create: POST /posts/
27 // TODO: Add a "Create" action
28
29 // Update: PUT /posts/:postId/
30 // TODO: Add an "Update" action
31
32 module.exports = router;
33
```

4. ROUTES AND CONTROLLERS

Our REST API will act as an interface to blog posts stored in a relational database. As such, we will be defining the following actions for accessing and manipulating posts:

HTTP Method	Path	Name	Description
GET	/posts	Index	List posts
POST	/posts	Create	Create new post
GET	/posts/:postId	Show	Show single post

<i>DELETE</i>	/posts/:postId	Destroy	Destroy existing post
<i>PUT</i>	/posts/:postId	Update	Update existing post

It is good programming practice to group related actions together in the same controller. In our case, this means that all of the actions which apply to posts should be grouped into a posts controller. Such a controller already exists in the file **blog/api/src/controllers/posts.js**. If you open up that file you will notice that three endpoints have been defined (index, show, and destroy actions). These endpoints are currently set up to answer requests with dummy responses.

4.1. CONTROLLER ROUTING

4.1.1. 404 NOT FOUND

Try to use HTTPie to send a request to the index endpoint (GET **/posts**):

```
http GET localhost:3001/posts
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ http GET localhost:3001/posts
HTTP/1.1 404 Not Found
Connection: keep-alive
Content-Length: 144
Content-Security-Policy: default-src 'self'
Content-Type: text/html; charset=utf-8
Date: Wed, 23 Aug 2023 09:49:07 GMT
Keep-Alive: timeout=5
X-Content-Type-Options: nosniff
X-Powered-By: Express
```

Oh no, “404 Not Found”! Don't worry, this is to be expected as we have not yet connected the posts controller to the **/posts** path. This process of associating controller actions with paths is known as routing.

4.1.2. CONNECT POSTS CONTROLLER

Edit **blog/api/src/config/routes.js** to connect the posts controller to **/posts**.

```
routes.connect = (app) => {
  app.use('/', index);
  app.use('/posts', posts) // <-- Insert this line
};
```

JS routes.js M X

```
api > src > config > JS routes.js > ...
1  const index = require('../controllers/index');
2  const posts = require('../controllers/posts');
3
4  const routes = {};
5
6  // Connect our controllers to specific base paths.
7  // For example, actions defined in our posts controller should be available at
8  // paths beginning with /posts.
9  routes.connect = (app) => {
10   // Use the index controller for /
11   app.use('/', index);
12   // TODO: Use the posts controller for /posts
13   app.use('/posts', posts);
14 },
15
16 module.exports = routes;
17
```

4.1.3. TRY AGAIN

Use HTTPie to once again send a request to the index endpoint (4.1.1). Now your response should look like:

```
HTTP/1.1 200 OK
...
{
  "todo": "List posts"
}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ http GET localhost:3001/posts
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 21
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:51:29 GMT
ETag: W/"15-Sv70HnrfWg1qnZvSZ/j1dJv1FoQ"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "todo": "List posts"
}
```

4.2. HTTP METHODS

4.2.1. RETURN

Return to the posts controller in your text editor.

4.2.2. PAY ATTENTION

Pay particular attention to the show action and destroy action. Although the relative path is the same (`/:postId`), they respond to different HTTP methods:

```
router.get('/:postId') // <- Show: GET /posts/:postId
router.delete('/:postId') // <- Destroy: DELETE /posts/:postId
```



```

9
10 // Show: GET /posts/:postId/
11 //
12 // Note that the path contains a variable (the :postId part). This will be
13 // made available as a property of the req.params object.
14 router.get('/:postId', (req, res) => {
15 |   res.json({ todo: 'Show post with ID=' + req.params.postId });
16 | });
17
18 // Destroy: DELETE /posts/:postId/
19 //
20 // Note that the path is the same as the "Show" action, but the HTTP method
21 // is different (we are using router.delete instead of router.get).
22 router.delete('/:postId', (req, res) => {
23 |   res.json({ todo: 'Delete post with ID=' + req.params.postId });
24 | });
25

```

In Express, **router.get** responds to GET requests, and **router.delete** responds to DELETE requests. Similar router methods exist for PUT and POST.

4.2.3. TWO REQUESTS

Use HTTPie to send two requests to **/posts/13**, first using GET and then DELETE:

```
http GET localhost:3001/posts/13
```

```

vboxuser@CSE5006:~/Documents/lab-7/blog$ http GET localhost:3001/posts/13
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 31
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:55:49 GMT
ETag: W/"1f-ca1xzK8m3dPsZIGa/NqVqE32bKQ"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "todo": "Show post with ID=13"
}

```

```
http DELETE localhost:3001/posts/13
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ http DELETE localhost:3001/posts/13
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 33
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 09:56:24 GMT
ETag: W/"21-Yq1kWjdxOpAJwOpskuo1kgn0rBU"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "todo": "Delete post with ID=13"
}
```

You should notice that the response changes depending on the HTTP method, even though the URL is the same for both requests.

4.2.4. EXERCISE 1

As mentioned earlier, three endpoints have been defined already (index, show, and destroy). Using these as a reference, add the remaining two actions (create and update). Refer to the table from earlier in this section to determine the path and HTTP method for each endpoint. You do not need to access the database yet, just respond with an appropriate placeholder message like the other existing actions.

Use HTTPie to test both endpoints. For example, a successful test of the update action would look like:

```
http PUT localhost:3001/posts/1
```

```
HTTP/1.1 200 OK
...
{
  "todo": "Update post with ID=1"
}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ http POST localhost:3001/posts/
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 24
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 10:00:53 GMT
ETag: W/"18-icjIjv8xrk2JnRZX5el50eydzkA"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "todo": "Insert new ID"
}

vboxuser@CSE5006:~/Documents/lab-7/blog$ http PUT localhost:3001/posts/13
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 33
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 10:00:56 GMT
ETag: W/"21-IB6mbW8p0oupP/XJZpWsXavPUro"
Keep-Alive: timeout=5
X-Powered-By: Express

{
  "todo": "Update post with ID=13"
}
```

5. CREATING AND LINKING A DATABASE CONTAINER

Hopefully the previous section illustrated how easy it is to create a REST API in Node.js. Of course, the API doesn't actually do anything useful, and this is where the real work comes in. For our application we want to be able to access and modify blog posts, so we will need some kind of persistent storage for blog post data. A popular option for storage is to use a relational SQL database. Relational databases have a long history of research, have matured after years of heavy industry use, and are both fast and reliable. For these reasons we will be using a relational database, MySQL, for data storage.

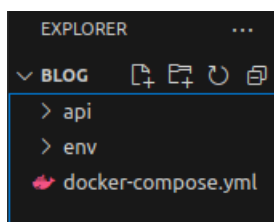
In order to keep our project modular and scalable, we will be including MySQL as a separate container as opposed to adding it to the API image itself. Fortunately, there is an existing MySQL image from a company called Tutum which we can use right out of the box. You can read about it on Docker Hub.

5.1. ENVIRONMENT VARIABLE CONFIGURATION

Firstly we will write a configuration file containing the administration username and password for the database, which will be provided to our containers as environment variables.

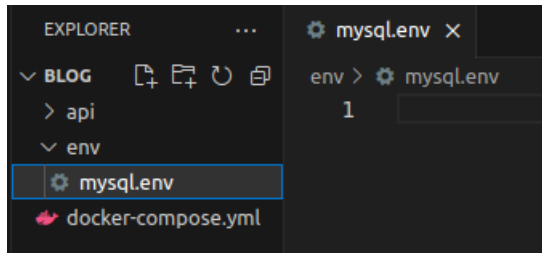
5.1.1. NEW FOLDER

Create a new folder in the **blog/** project directory root called **env**.



5.1.2. NEW FILE

Inside this folder create a file called **mysql.env**.



5.1.3. RANDOM PASSWORD

Generate a secure random password and copy it to the clipboard.

You may use this function to create a very complex password, or define your own simple password.

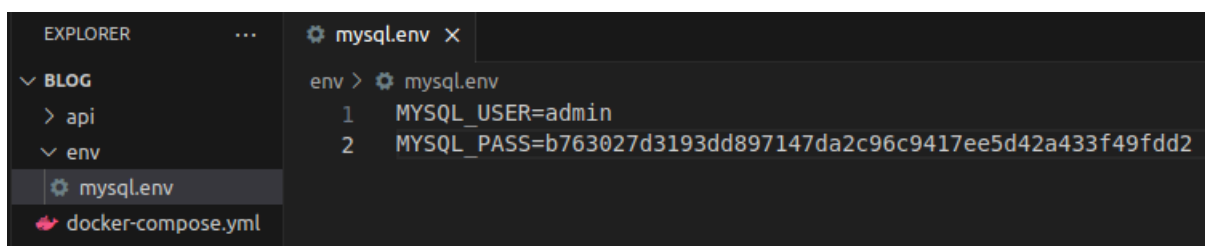
```
openssl rand -hex 24
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog$ openssl rand -hex 24
b763027d3193dd897147da2c96c9417ee5d42a433f49fdd2
vboxuser@CSE5006:~/Documents/lab-7/blog$
```

5.1.4. EDIT AND PAST

Now edit **mysql.env** to contain the following (pasting your random password where indicated):

```
MYSQL_USER=admin
MYSQL_PASS=<pasted_password>
```



5.2. DOCKER COMPOSE SERVICE

5.2.1. STOP

If you still have it running, stop Docker compose with **Ctrl+C**.

5.2.2. NEW SERVICE

Add a new service to **docker-compose.yml** so that we can actually use the Tutum MySQL image to run our database instance. Make sure you put the right indentation here:

```
version: "3.8"

services:
  api:
    build: api
    environment:
      - NODE_ENV=development
    volumes:
      - "./api:/app"
    env_file:
      - ./env/mysql.env
    ports:
      - "3001:3000"
    links:
      - db

  db:
    image: tutum/mysql:5.6
    environment:
      - ON_CREATE_DB=development_db
    env_file:
      - ./env/mysql.env
    volumes:
      - "blog-db-data:/var/lib/mysql"

volumes:
  blog-db-data:
    external: false
```

There's quite a bit going on here, so I'll step you through some key points. Firstly, we have added a new service called **db** which, instead of building from a Dockerfile, will be created from the image **tutum/mysql**. This service has the environment variable **ON_CREATE_DB** set, which tells the MySQL container to create a database with the name "development_db". We also mount a volume at the path **/var/lib/mysql** in the container, which is where all of the MySQL database data will be stored. We have also added a link to the db service from the api service. This link tells Docker compose that the api service requires the db service, and will also set environment variable inside the API container specifying the IP address and port on which the database is running. More on this later. Finally, we have added an **env_file** field to both the api and db services. This field tells Docker compose to read the file **env/mysql.env** and set environment variables according to its contents - therefore both services will have access to the admin login details for the database.

5.2.3. TRY IT

Great! Now when we use Docker compose up we will be starting both our API container and a configured MySQL container. Try it, the output should show messages from both api and db.

```
blog-db-1 | =====
blog-db-1 | You can now connect to this MySQL Server using:
blog-db-1 |
blog-db-1 |      mysql -uadmin -pb763027d3193dd897147da2c96c9417ee5d42a433f49fdd2 -h<host> -P<port>
blog-db-1 |
blog-db-1 | Please remember to change the above password as soon as possible!
blog-db-1 | MySQL user 'root' has no password but only allows local connections
blog-db-1 | =====
blog-db-1 | Creating MySQL database development_db
blog-db-1 | Database created!
blog-db-1 | tail -F $LOG
```

6. USING THE DATABASE IN NODE

Add the following lines to **mysql.env**, which tell the api service how to reach the database service:

```
MYSQL_REMOTE_HOST=db
MYSQL_REMOTE_PORT=3306
```

```
mysql.env u
env > mysql.env
1  MYSQL_USER=admin
2  MYSQL_PASS=8b9779263a2f0c5ab25ed2201c74f08ff8546b17b7f9e306
3  MYSQL_REMOTE_HOST=db
4  MYSQL_REMOTE_PORT=3306
```

Here's a complete list of the database-related environment variables:

Variable	Description
<code>MYSQL_USER</code>	Database admin username
<code>MYSQL_PASS</code>	Database admin password
<code>MYSQL_REMOTE_HOST</code>	The hostname of the database service
<code>MYSQL_REMOTE_PORT</code>	The port of the database service

This means that our Node.js REST now knows where to find the database and how to log in. To make life easier we will be using an ORM (object-relational mapping) library called Sequelize to interface with the database. The nice thing about Sequelize is that it takes care of running database queries and converting to and from JavaScript objects for us. Sequelize has already been included for you in the project dependencies, so let's get started right away.

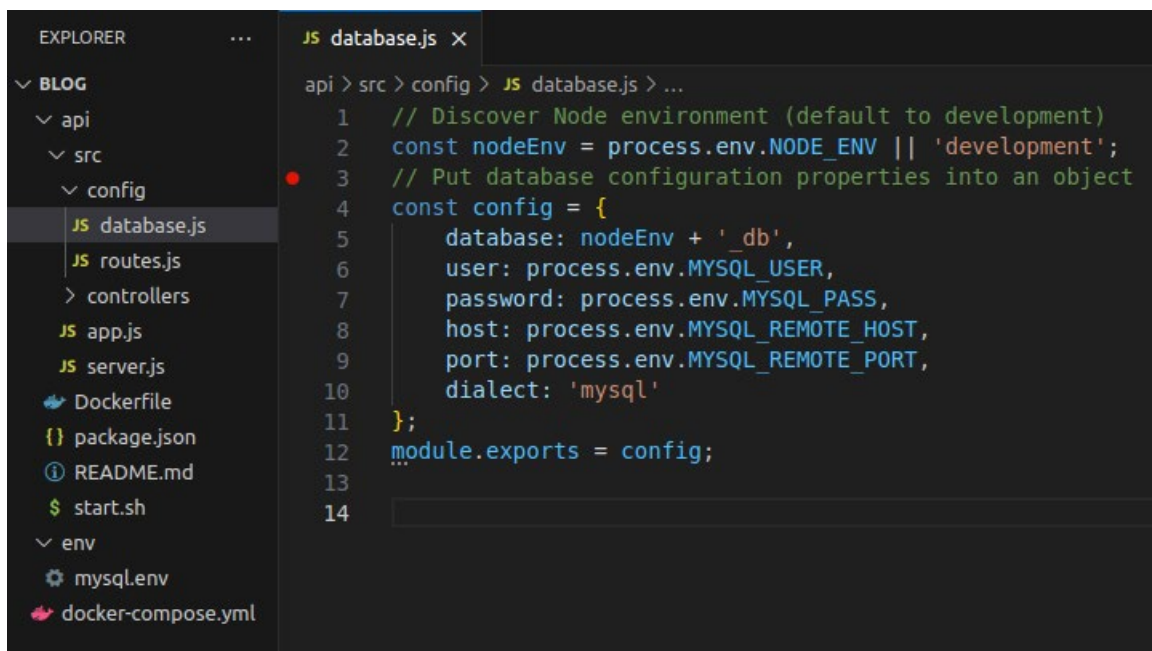
6.1. CONFIGURING THE DATABASE CONNECTION

We are now going to configure our project to use Sequelize with the database location and login credentials from the environment variables we just defined.

6.1.1. NEW FILE

Create a new file, **blog/api/src/config/database.js**, which will create a JavaScript object containing all of the connection details required by Sequelize.

```
// Discover Node environment (default to development)
const nodeEnv = process.env.NODE_ENV || 'development';
// Put database configuration properties into an object
const config = {
  database: nodeEnv + '_db',
  user: process.env.MYSQL_USER,
  password: process.env.MYSQL_PASS,
  host: process.env.MYSQL_REMOTE_HOST,
  port: process.env.MYSQL_REMOTE_PORT,
  dialect: 'mysql'
};
module.exports = config;
```



6.1.2. ANOTHER NEW FILE

Create another new file called **.sequelizerc**, this time in the root of the API project directory (alongside the Dockerfile and so forth). This file is read by the Sequelize command line tool to configure the database connection and other options. The contents of **.sequelizerc** are as follows:


```

const path = require('path');
const dbConfig = require('./src/config/database');
// Build the connection URL string
const connectionUrl = 'mysql://' +
    dbConfig.user + ':' + dbConfig.password + '@' +
    dbConfig.host + ':' + dbConfig.port + '/' + dbConfig.database;
// Export settings for the Sequelize command line tool
module.exports = {
    'url': connectionUrl,
    'migrations-path': path.resolve('src', 'migrations'),
    'models-path': path.resolve('src', 'models'),
    'seeders-path': path.resolve('src', 'seeders')
};

```

```

api > JS .sequelizerc > ...
1
2 const path = require('path');
3 const dbConfig = require('./src/config/database');
4 // Build the connection URL string
5 const connectionUrl = 'mysql://' +
6     dbConfig.user + ':' + dbConfig.password + '@' +
7     dbConfig.host + ':' + dbConfig.port + '/' + dbConfig.database;
8 // Export settings for the Sequelize command line tool
9 module.exports = {
10     'url': connectionUrl,
11     'migrations-path': path.resolve('src', 'migrations'),
12     'models-path': path.resolve('src', 'models'),
13     'seeders-path': path.resolve('src', 'seeders')
14 };
15

```

Notice how we load our database configuration from **blog/api/src/config/database.js** and use it to create a connection string for Sequelize. We also specify that we want to store migrations, models, and seeders under the **blog/api/src/** directory (more on these shortly).

make sure you can see your file in /api folder by running **\$ ls -l -a**

```

vboxuser@CSE5006:~/Documents/lab-7/blog/api$ ls -l -a
total 32
drwxrwxr-x 3 vboxuser vboxuser 4096 Aug 23 20:17 .
drwxrwxr-x 4 vboxuser vboxuser 4096 Aug 23 20:04 ..
-rw-rw-r-- 1 vboxuser vboxuser 1291 Aug 23 19:21 Dockerfile
-rw-rw-r-- 1 vboxuser vboxuser 373 Aug 23 19:21 package.json
-rw-rw-r-- 1 vboxuser vboxuser 86 Aug 23 19:21 README.md
-rw-rw-r-- 1 vboxuser vboxuser 534 Aug 23 20:19 .sequelizerc
drwxrwxr-x 4 vboxuser vboxuser 4096 Aug 23 19:21 src
-rwxr-w-r-- 1 vboxuser vboxuser 653 Aug 23 19:21 start.sh
vboxuser@CSE5006:~/Documents/lab-7/blog/api$

```

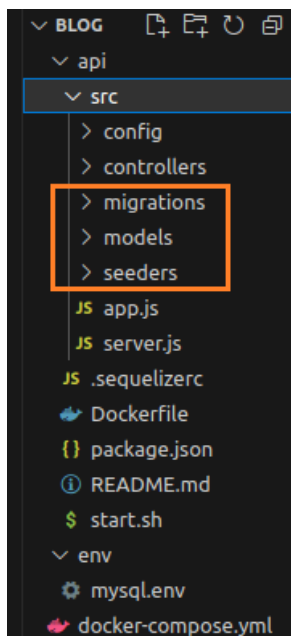
6.1.3. CREATE SUBDIRECTORIES

Run the following command from the **blog/api/** folder to actually create these subdirectories:

```
mkdir -p src/{models,migrations,seeders}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ mkdir -p src/{models,migrations,seeders}
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

Please verify if you get these three folders correctly



6.2. CREATING A MODEL

With the database connection correctly set up we can now use the sequelize command to generate a database table for blog posts. This command will be run inside the container, but will produce files that should become part of the project. Since our project directory is mounted inside the container, the files created by the sequelize command will appear on the host file system.

6.2.1. RUN THE COMMAND

Using docker-compose run we can run the sequelize command in the API service to define a Post model with two attributes - **title** and **content**.

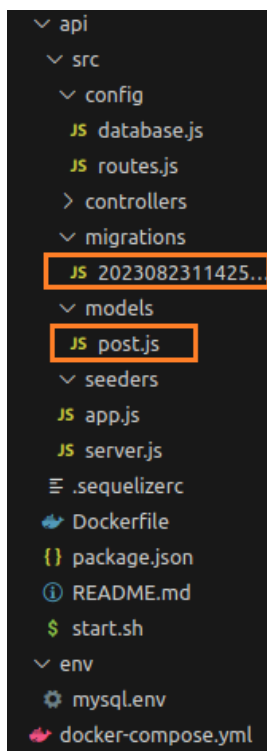
```
docker compose run --rm api sequelize model:create \
  --name Post --attributes title:string,content:text
```

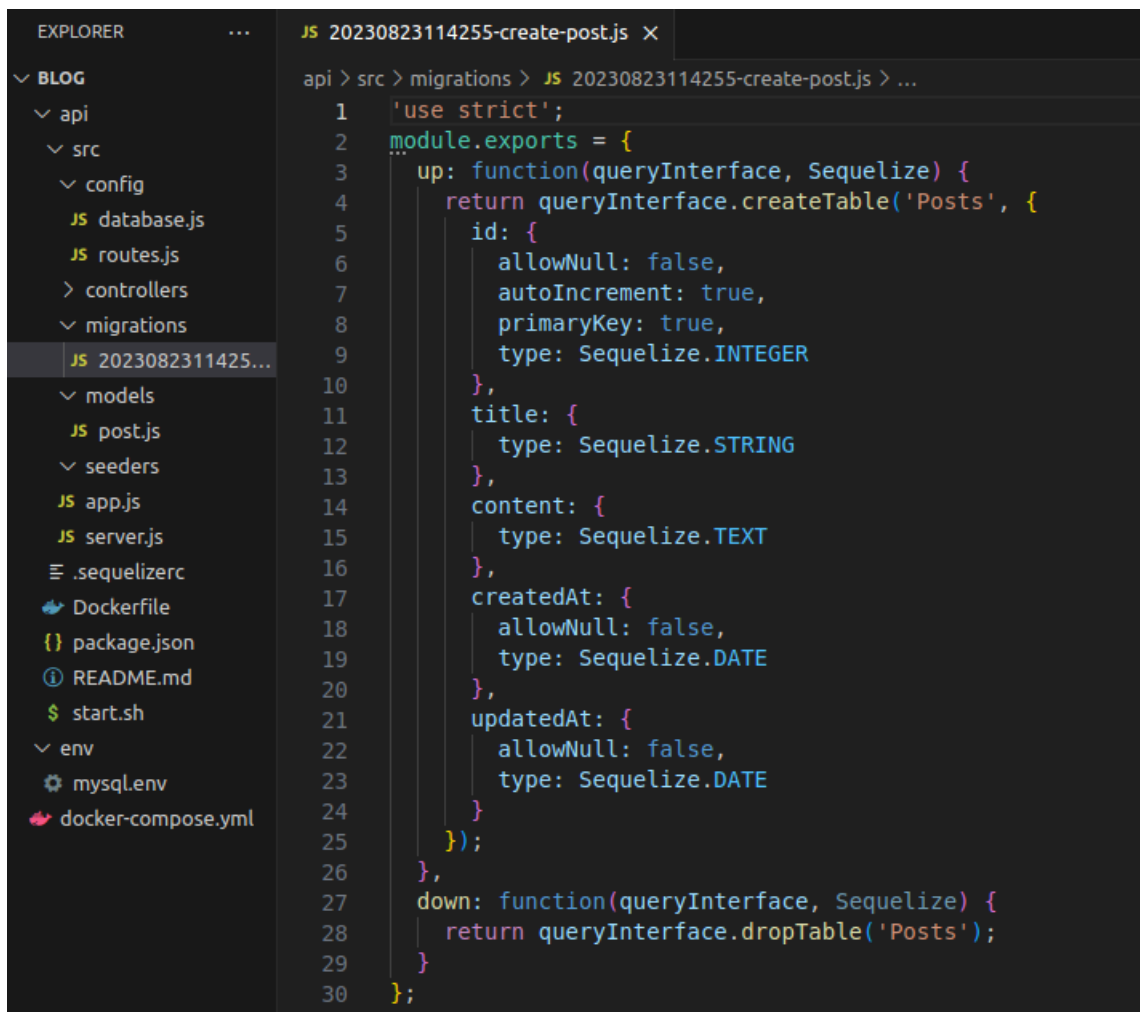
```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ docker compose run --rm api sequelize model:create --name Post --attributes title:string,content:text
[+] Creating 1/0
✓ Container blog-db-1 Running 0.0s

Sequelize [Node: 8.1.2, CLI: 2.7.0, ORM: 3.30.4]

Parsed url mysql://admin:*****@db:3306/development_db
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

If the command doesn't work, double check that you have saved **.sequelizerc** in the right place (it must be in the **api/** folder). When the command runs successfully, two new files will be created. One of these files describes how to create the database table itself, and is called a "migration". This file is located in the **src/migrations** directory with the name **<timestamp>-create-post.js**. Open this file up in VS Code and take a look. Notice that apart from the title and content attributes, some other useful fields have been added for us. One of these is **id**, which acts as a unique primary key for identifying the post. The other two attributes are used to record timestamp information for creation and modification events. These automatically added fields have been set to disallow null values by setting **allowNull** to false.



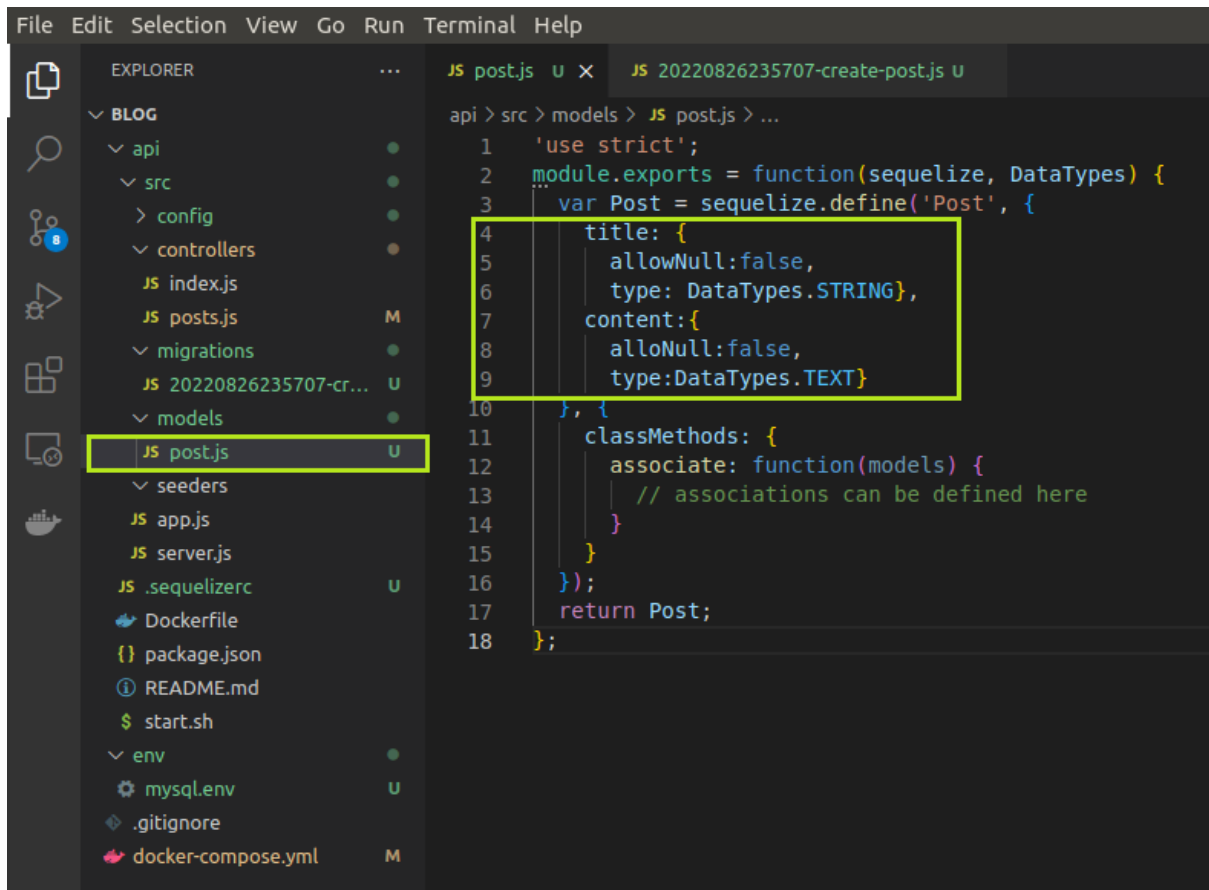


The image shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure for a 'BLOG' application. The code editor displays a JavaScript file named '20230823114255-create-post.js' with the following content:

```
1 'use strict';
2 module.exports = {
3   up: function(queryInterface, Sequelize) {
4     return queryInterface.createTable('Posts', {
5       id: {
6         allowNull: false,
7         autoIncrement: true,
8         primaryKey: true,
9         type: Sequelize.INTEGER
10      },
11      title: {
12        type: Sequelize.STRING
13      },
14      content: {
15        type: Sequelize.TEXT
16      },
17      createdAt: {
18        allowNull: false,
19        type: Sequelize.DATE
20      },
21      updatedAt: {
22        allowNull: false,
23        type: Sequelize.DATE
24      }
25    });
26  },
27  down: function(queryInterface, Sequelize) {
28    return queryInterface.dropTable('Posts');
29  }
30 };
```

6.2.2. MODIFY TITLE AND CONTENT

Modify the title and content attributes for the Post model so they also prevent null values.



6.2.3. ADD TABLE

To actually add the table to the database, we need to run the migration. This can be achieved using another sequelize command.

```
docker compose run --rm api sequelize db:migrate
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ docker compose run --rm api sequelize db:migrate
[+] Creating 1/0
✓ Container blog-db-1 Running

Sequelize [Node: 8.1.2, CLI: 2.7.0, ORM: 3.30.4]

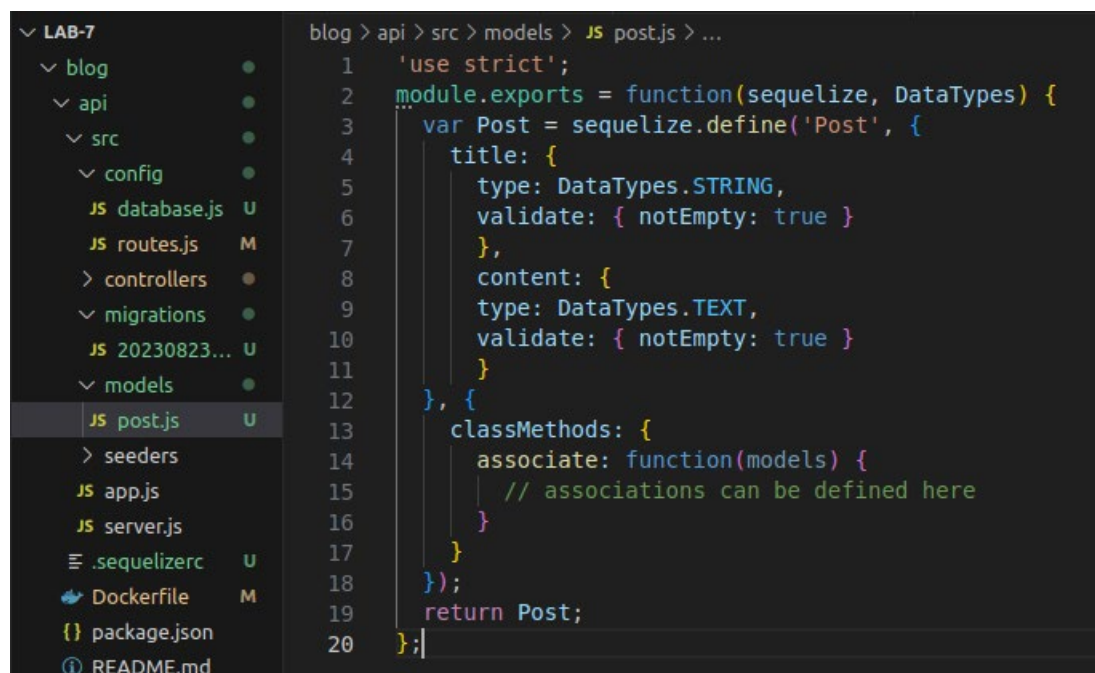
Parsed url mysql://admin:*****@db:3306/development_db
== 20230823114255-create-post: migrating =====
== 20230823114255-create-post: migrated (0.080s)
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

This will go through the files in **src/migrations** and run each in order. In our case there is only one file, which will create the database table for blog posts.

6.2.4. DEFINITION

Besides creating a migration, the **sequelize model:create** command we ran earlier also created a model definition (**src/models/post.js**). The model definition provides Sequelize with higher-level details about the model which allows it to map between raw database records and JavaScript objects more effectively. For example, we can specify validations to enforce certain facts about a record before it can be stored in the database. We will add one such validation which checks that neither the title nor the content of a post are empty.

```
title: {
  type: DataTypes.STRING,
  validate: { notEmpty: true }
},
content: {
  type: DataTypes.TEXT,
  validate: { notEmpty: true }
}
```



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'blog', 'api', 'src', 'config', 'controllers', 'migrations', 'models', 'seeders', and files like 'app.js', 'server.js', '.sequelizerc', 'Dockerfile', 'package.json', and 'README.md'. The 'models' folder is expanded, showing 'post.js'. The code editor on the right shows the content of 'post.js' with line numbers 1 through 20. The code defines a Sequelize model for 'Post' with validations for 'title' and 'content' to be non-empty, and an 'associate' method for defining associations.

```
blog > api > src > models > JS post.js > ...
1  'use strict';
2  module.exports = function(sequelize, DataTypes) {
3    var Post = sequelize.define('Post', {
4      title: {
5        type: DataTypes.STRING,
6        validate: { notEmpty: true }
7      },
8      content: {
9        type: DataTypes.TEXT,
10       validate: { notEmpty: true }
11      }
12    }, {
13      classMethods: {
14        associate: function(models) {
15          // associations can be defined here
16        }
17      }
18    });
19    return Post;
20  };
```

6.3. CREATING A SEEDER

Running migrations to create the database structure is handy, but testing during development is difficult with an empty database. To get around this, we can define dummy data to populate the database with using a seeder.

6.3.1. CREATE SEEDER

Use the Sequelize command to create a seeder for the Post records:

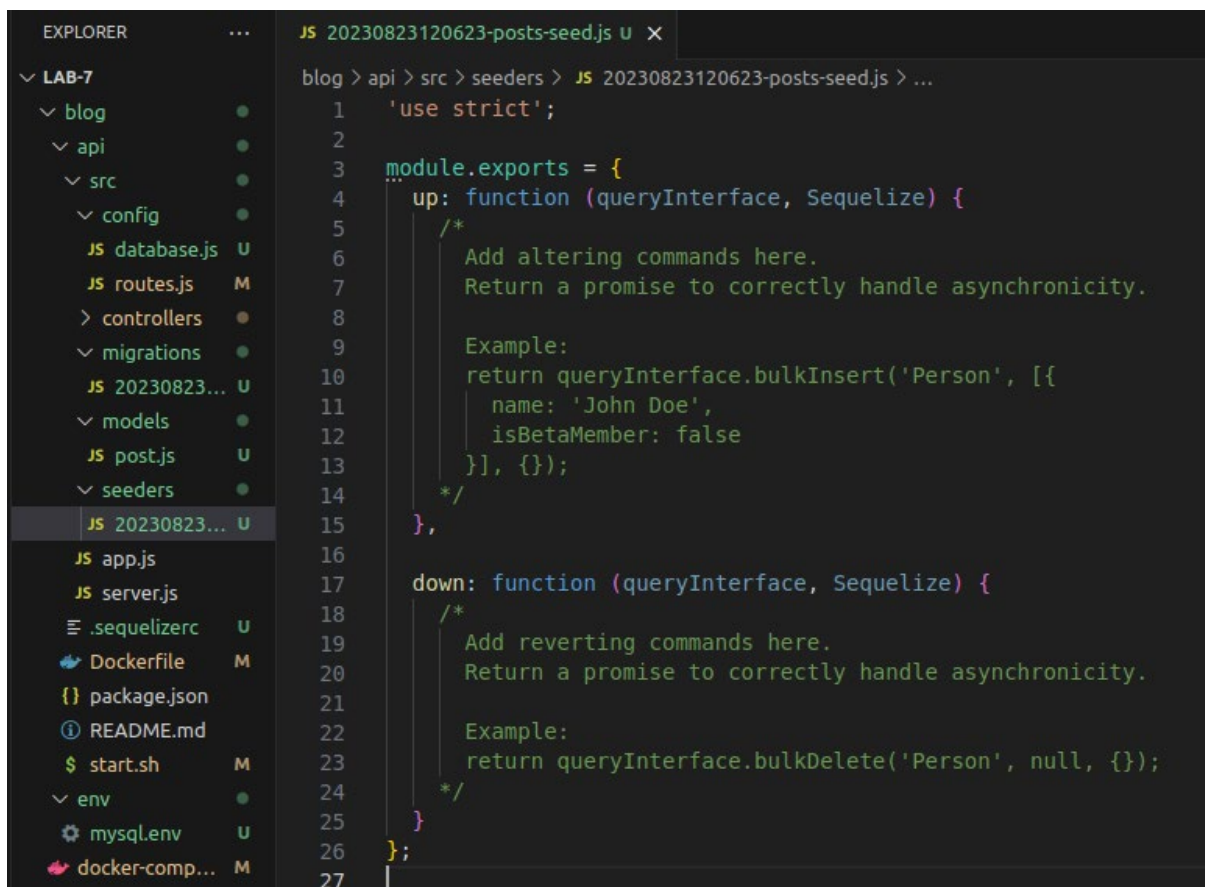
```
docker compose run --rm api sequelize seed:create \
  --name posts-seed
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ docker compose run --rm api sequelize seed:create --name posts-seed
[+] Creating 1/0
✓ Container blog-db-1 Running

Sequelize [Node: 8.1.2, CLI: 2.7.0, ORM: 3.30.4]

Successfully created seeders folder at "/app/src/seaders".
Parsed url mysql://admin:*****@db:3306/development_db
New seed was created at /app/src/seaders/20230823120623-posts-seed.js .
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

This will create a new file in **src/seaders**, which you should open up in VS Code now.



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows the project structure under 'LAB-7':

- blog
 - api
 - src
 - config
 - database.js (U)
 - routes.js (M)
 - controllers (●)
 - migrations (●)
 - 20230823... (U)
 - models (●)
 - post.js (U)
 - seaders (●)
- app.js
- server.js
- .sequelizerc (U)
- Dockerfile (M)
- package.json
- README.md
- start.sh (M)
- env
 - mysql.env (U)
- docker-comp... (M)

The main editor shows the file `20230823120623-posts-seed.js` with the following content:

```
blog > api > src > seaders > JS 20230823120623-posts-seed.js > ...
1  'use strict';
2
3  module.exports = {
4    up: function (queryInterface, Sequelize) {
5      /*
6       * Add altering commands here.
7       * Return a promise to correctly handle asynchronicity.
8       *
9       * Example:
10      return queryInterface.bulkInsert('Person', [{
11        name: 'John Doe',
12        isBetaMember: false
13      }], {});
14      */
15    },
16
17    down: function (queryInterface, Sequelize) {
18      /*
19       * Add reverting commands here.
20       * Return a promise to correctly handle asynchronicity.
21       *
22       * Example:
23      return queryInterface.bulkDelete('Person', null, {});
24      */
25    }
26  };
27
```

6.3.2. UP FUNCTION

Complete the up function in the seeder so that it creates 6 dummy records, using the code provided below as a starting point (replace the existing file contents).


```

module.exports = {
  up: function(queryInterface, Sequelize) {
    // Define dummy data for posts
    const posts = [
      {
        title: 'A silly post',
        content: 'Roses are red, violets are blue, I am a poet.',
        createdAt: new Date('2010/08/17 12:09'),
        updatedAt: new Date('2010/08/17 12:09')
      },
      {
        title: 'New technology',
        content: 'These things called "computers" are fancy.',
        createdAt: new Date('2011/03/06 15:32'),
        updatedAt: new Date('2011/03/06 15:47')
      }
    ];
    // Insert posts into the database
    return queryInterface.bulkInsert('Posts', posts, {});
  },
  down: function(queryInterface, Sequelize) {
    // Delete all posts from the database
    return queryInterface.bulkDelete('Posts', null, {});
  }
};

```

```

EXPLORER  ...  JS 20230823120623-posts-seed.js U X
blog > api > src > seeders > JS 20230823120623-posts-seed.js > ...
1  'use strict';
2
3  module.exports = {
4    up: function(queryInterface, Sequelize) {
5      // Define dummy data for posts
6      const posts = [
7        {
8          title: 'A silly post',
9          content: 'Roses are red, violets are blue, I am a poet.',
10         createdAt: new Date('2010/08/17 12:09'),
11         updatedAt: new Date('2010/08/17 12:09')
12       },
13       {
14         title: 'New technology',
15         content: 'These things called "computers" are fancy.',
16         createdAt: new Date('2011/03/06 15:32'),
17         updatedAt: new Date('2011/03/06 15:47')
18       }
19     ];
20     // Insert posts into the database
21     return queryInterface.bulkInsert('Posts', posts, {});
22   },
23   down: function(queryInterface, Sequelize) {
24     // Delete all posts from the database
25     return queryInterface.bulkDelete('Posts', null, {});
26   }
27 };

```

6.3.3. SEED DUMMY RECORDS

Once you are happy with the seeder, you can actually seed the database with your dummy records.

```
docker compose run --rm api sequelize db:seed:all
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ docker compose run --rm api sequelize db:seed:all
[+] Creating 1/0
✓ Container blog-db-1 Running

Sequelize [Node: 8.1.2, CLI: 2.7.0, ORM: 3.30.4]

Parsed url mysql://admin:*****@db:3306/development_db
== 20230823120623-posts-seed: migrating =====
== 20230823120623-posts-seed: migrated (0.017s)
vboxuser@CSE5006:~/Documents/lab-7/blog/api$
```

You can check if the data has been successfully inserted to the database by using MySQL console. Open a new terminal and execute the following scripts

```
$ docker exec -it blog-db-1 bash
```

```
$ mysql --user=$MYSQL_USER --password=$MYSQL_PASS development_db
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ docker exec -it blog-db-1 bash
root@8c4390049307:/# mysql --user=$MYSQL_USER --password=$MYSQL_PASS development_db
Warning: Using a password on the command line interface can be insecure.
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 5.6.28-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show tables;
+-----+
| Tables_in_development_db |
+-----+
| Posts                     |
| SequelizeMeta             |
+-----+
2 rows in set (0.00 sec)

mysql> select * from Posts;
+-----+-----+-----+-----+
| id | title           | content                                     | createdAt          | updatedAt          |
+-----+-----+-----+-----+
| 1  | A silly post    | Roses are red, violets are blue, I am a poet. | 2010-08-17 12:09:00 | 2010-08-17 12:09:00 |
| 2  | New technology  | These things called "computers" are fancy.   | 2011-03-06 15:32:00 | 2011-03-06 15:47:00 |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Type “exit” twice to return to the Ubuntu terminal.

Notes: in this example, the microservice name is blog-db-1. You may have different name. Check it when running the docker containers using docker compose.

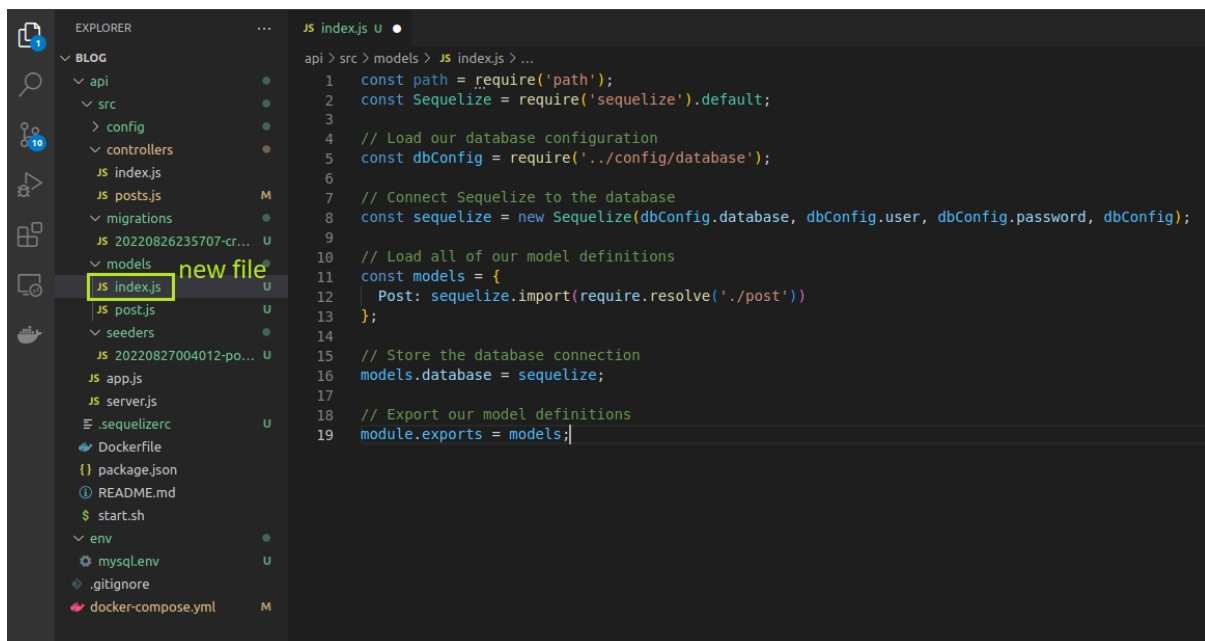
6.4. USING THE MODEL FROM THE CONTROLLER

We're almost there, I promise! So far we've created a database table and filled it with some dummy data. We've also created a model file which Sequelize can use to map raw database records to JavaScript objects. All that's left for us to do is initialise Sequelize with our database connection details, and load our model definition.

6.4.1. CREATE CONTENTS

We will do this in the `src/models/index.js` file, so create it now with the following contents:

```
const Sequelize = require('sequelize');
// Load our database configuration
const dbConfig = require('../config/database');
// Connect Sequelize to the database
const sequelize = new Sequelize(dbConfig.database, dbConfig.user,
dbConfig.password, dbConfig);
// Load all of our model definitions
const models = {
  Post: sequelize.import(require.resolve('./post'))
};
// Store the database connection
models.database = sequelize;
// Export our model definitions
module.exports = models;
```



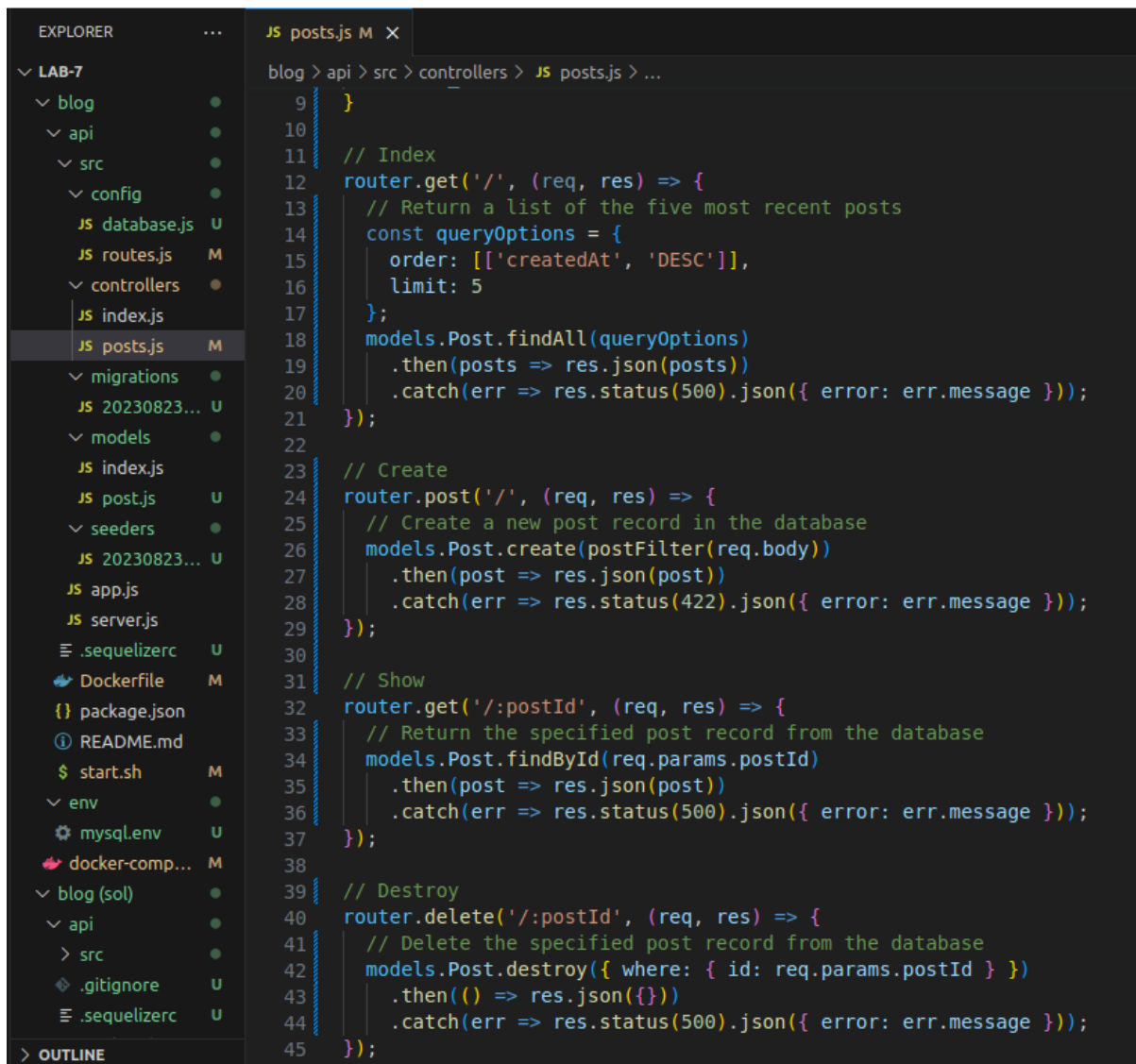
6.4.2. POST MODEL

Now we can finally use the Post model! Open up the posts controller (`src/controllers/posts.js`) and modify it to actually perform database transactions.

```

const express = require('express');
const _ = require('lodash');
const models = require('../models');
const router = express.Router();
// Selects only the fields that are allowed to be set by users
function postFilter(obj) {
  return _.pick(obj, ['title', 'content']);
}
// Index
router.get('/', (req, res) => {
  // Return a list of the five most recent posts
  const queryOptions = {
    order: [['createdAt', 'DESC']],
    limit: 5
  };
  models.Post.findAll(queryOptions)
    .then(posts => res.json(posts))
    .catch(err => res.status(500).json({ error: err.message }));
});
// Create
router.post('/', (req, res) => {
  // Create a new post record in the database
  models.Post.create(postFilter(req.body))
    .then(post => res.json(post))
    .catch(err => res.status(422).json({ error: err.message }));
});
// Show
router.get('/:postId', (req, res) => {
  // Return the specified post record from the database
  models.Post.findById(req.params.postId)
    .then(post => res.json(post))
    .catch(err => res.status(500).json({ error: err.message }));
});
// Destroy
router.delete('/:postId', (req, res) => {
  // Delete the specified post record from the database
  models.Post.destroy({ where: { id: req.params.postId } })
    .then(() => res.json({}))
    .catch(err => res.status(500).json({ error: err.message }));
});
// Update
// TODO: Implement the update action here
module.exports = router;

```



```
9 }
10
11 // Index
12 router.get('/', (req, res) => {
13   // Return a list of the five most recent posts
14   const queryOptions = {
15     order: [['createdAt', 'DESC']],
16     limit: 5
17   };
18   models.Post.findAll(queryOptions)
19     .then(posts => res.json(posts))
20     .catch(err => res.status(500).json({ error: err.message }));
21 });
22
23 // Create
24 router.post('/', (req, res) => {
25   // Create a new post record in the database
26   models.Post.create(postFilter(req.body))
27     .then(post => res.json(post))
28     .catch(err => res.status(422).json({ error: err.message }));
29 });
30
31 // Show
32 router.get('/:postId', (req, res) => {
33   // Return the specified post record from the database
34   models.Post.findById(req.params.postId)
35     .then(post => res.json(post))
36     .catch(err => res.status(500).json({ error: err.message }));
37 });
38
39 // Destroy
40 router.delete('/:postId', (req, res) => {
41   // Delete the specified post record from the database
42   models.Post.destroy({ where: { id: req.params.postId } })
43     .then(() => res.json({}))
44     .catch(err => res.status(500).json({ error: err.message }));
45 });
```

6.4.3. COMMIT

Make sure that you add, commit and push your changes using Git.

6.4.4. EXERCISE 2

Implement the REST endpoint PUT **/posts/:id** (the update action) by creating a new action in the posts controller to update the attributes of a post. This will involve finding the post by its ID, then using the **update()** method on the found post. Refer to the “show” and “create” actions for help, as “update” uses a bit of code from both.

6.4.5. EXERCISE 3

Use HTTPie to test the REST API (refer back to Section 2 if you need to). You shall do this by sending the requests described below (in order):

6.4.5.1 LIST ALL POSTS

List all posts. I'll give you the HTTPie command for this one to get you started:

```
http GET localhost:3001/posts
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http GET localhost:3001/posts
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 338
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:31:10 GMT
ETag: W/"152-5SCdi1BBE9iLWqgtF8h1lN+FwGQ"
X-Powered-By: Express

[
  {
    "content": "These things called \"computers\" are fancy.",
    "createdAt": "2011-03-06T15:32:00.000Z",
    "id": 2,
    "title": "New technology",
    "updatedAt": "2011-03-06T15:47:00.000Z"
  },
  {
    "content": "Roses are red, violets are blue, I am a poet.",
    "createdAt": "2010-08-17T12:09:00.000Z",
    "id": 1,
    "title": "A silly post",
    "updatedAt": "2010-08-17T12:09:00.000Z"
  }
]
```

6.4.5.2. CREATE NEW POST

Create a new post with title="WDC" and content="World domination cat". Make a note of the returned ID since it is required for the next few actions.

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http POST
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 133
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:36:10 GMT
ETag: W/"85-5BT1kaqsX+rMyMZZNN4X6GkdKVA"
X-Powered-By: Express
```

```
{
  "content": "World domination cat",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:36:10.000Z"
}
```

6.4.5.3. UPDATE POST

Update the post so that the content is changed to “We dislike chess”. Use the post ID from step 2.

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 129
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:38:05 GMT
ETag: W/"81-8H3xfkVWKPL05q3rkIFVnry5c54"
X-Powered-By: Express
```

```
{
  "content": "We dislike chess",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:38:05.000Z"
}
```

6.4.5.4. SHOW POST

Show the post using the post ID from step 2. The content field in the response should be “We dislike chess”.

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 129
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:39:29 GMT
ETag: W/"81-8H3xfkVWKPL05q3rkIFVnry5c54"
X-Powered-By: Express
```

```
{
  "content": "We dislike chess",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:38:05.000Z"
}
```

http D

6.4.5.5. DELETE POST

Delete the post using the post ID from step 2.


```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 2
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:41:18 GMT
ETag: W/"2-vyGp6PvFo4RvsFtPoIWeCReyIC8"
X-Powered-By: Express

{}
```

http

6.4.5.6. LIST ALL POSTS

List all posts. The “WDC” note should not be present since it was deleted.

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http GET localhost:3001/posts
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 338
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:42:34 GMT
ETag: W/"152-5SCdi1BBE9iLWqgtF8h1lN+FwGQ"
X-Powered-By: Express

[
  {
    "content": "These things called \"computers\" are fancy.",
    "createdAt": "2011-03-06T15:32:00.000Z",
    "id": 2,
    "title": "New technology",
    "updatedAt": "2011-03-06T15:47:00.000Z"
  },
  {
    "content": "Roses are red, violets are blue, I am a poet.",
    "createdAt": "2010-08-17T12:09:00.000Z",
    "id": 1,
    "title": "A silly post",
    "updatedAt": "2010-08-17T12:09:00.000Z"
  }
]
```

Save the HTTPie command and response for each of the six requests to a file in the project directory. Don't forget to add, commit, and push that file to Bitbucket.

Answer:

Exercise 1

```
router.post('/', (req, res) => {
  // Create a new post record in the database
  res.json({todo: 'Insert new ID'});
});
```

```
router.put('/:postId', (req, res) => {
  res.json({todo: 'Update post with ID='+req.params.postId })
});
```

Exercise 2

```
// Update
// TODO: Implement the update action here
router.put('/:postId', (req, res) => {
  // Update the specified post record in the database
  models.Post.findById(req.params.postId)
    .then(post => post.update(postFilter(req.body)))
    .then(post => res.json(post))
    .catch(err => res.status(422).json({ error: err.message }));
});
```

Exercise 3

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http POST localhost:3001/posts title="WDC" content="World domination cat"
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 133
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:36:10 GMT
ETag: W/"85-5BTikaqsX+rMyMZZNN4X6GkdKVA"
X-Powered-By: Express

{
  "content": "World domination cat",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:36:10.000Z"
}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http PUT localhost:3001/posts/3 content="We dislike chess"
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 129
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:38:05 GMT
ETag: W/"81-8H3xfkVWKPL05q3rkIFVnry5c54"
X-Powered-By: Express

{
  "content": "We dislike chess",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:38:05.000Z"
}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http GET localhost:3001/posts/3
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 129
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:39:29 GMT
ETag: W/"81-8H3xfkVWKPL05q3rkIFVnry5c54"
X-Powered-By: Express

{
  "content": "We dislike chess",
  "createdAt": "2023-08-23T12:36:10.000Z",
  "id": 3,
  "title": "WDC",
  "updatedAt": "2023-08-23T12:38:05.000Z"
}
```

```
vboxuser@CSE5006:~/Documents/lab-7/blog/api$ http DELETE localhost:3001/posts/3
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 2
Content-Type: application/json; charset=utf-8
Date: Wed, 23 Aug 2023 12:41:18 GMT
ETag: W/"2-vyGp6PvFo4RvsFtPoIWeCReyIC8"
X-Powered-By: Express

{}
```