



CST-391 Activity 1 Guide

Contents

Part 1: Refining the Build	2
Overview	2
Scripts in package.json.....	2
Summary of Work.....	3
Part 2: Application Structure	4
Model-View-Controller (MVC).....	4
The Router	4
Middleware	9
Part 3: Connecting to the MySQL Database	12
MySQL Dependencies	12
MySQL Configuration	12
MySQL Connection Pool.....	13
Router Revisited.....	14
The Controller Learns to Communicate with MySQL	15
Reviewing the MusicAPI.....	27
Code Structure	27
RESTful API.....	27
Submission.....	28
Appendix: Implementation Notes	29
Complete Code Listing for <i>app.ts</i>	29
Supplemental Resources	30

Part 1: Refining the Build

Overview

In the previous activity, you created a simple Node.js web server, called MusicAPI, written in TypeScript. In this step, we will refine the build and execution of MusicAPI. Open your MusicAPI project and we will begin.

Scripts in package.json

1. Open package.json and make sure that the entry "main" is set to "app.ts". If you took the defaults when you first created package.json, it is probably set to "index.js":
`"main": "app.ts",`
2. In package.json, modify the "scripts" section. **Note:** you should find the "test" section already in package.json:
`"scripts": {
 "start:watch": "nodemon ./src/app.ts",
 "start": "npx ts-node ./src/app.ts",
 "test": "echo \"Error: no test specified\" && exit 1"
}`
3. With this modification, you should now be able to start your project from the root of your project folder with the command:
`npm run start`
or
`npm run start:watch`
4. Spend a little time looking at the script modification and the new "start" verb. In our Angular and React activities, you will be working with frameworks that have pre-built build scripts and sample source code. This step that we have done manually will help you understand how those systems work.
5. We also want to be able to use 'nodemon', which will restart our server when code modifications are made to our TypeScript files. This utility was installed in the earlier activity:
`npm install -g nodemon`
6. The TypeScript compiler needs a configuration file so that it will work properly in our application. In the root of your project, create a file called "tsconfig.json" with the following contents:

```
{  
  "compilerOptions": {  
    "target": "es2016",  
    "module": "commonjs",  
    "sourceMap": true,  
    "noEmitOnError": true,  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "skipLibCheck": true  
  }  
}
```

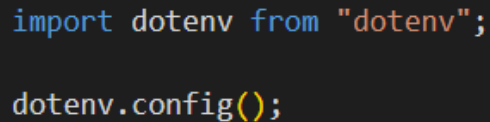
Figure 1 *tsconfig.json*

7. Environment Variable Utility

To use environment variables, you need to install and use the dotenv library:

```
npm install dotenv
```

In the 'app.ts' code file:



```
import dotenv from "dotenv";  
  
dotenv.config();
```

Figure 2 *Preparing Environment Variables*

dotenv

If environment variables fail to load, the path to the .env file needs to be specified:

```
dotenv.config({ path: "../.env" });
```

- a. The environment variables of a running program contain many settings inherited from the operating system. These are available to *any* executable and should be looked into by students in software development. To create local environment variables just for this application, create a '.env' (just an extension, no name) file in the root folder of your project with the following contents:

#MySQL Settings

MY_SQL_DB_HOST=127.0.0.1

MY_SQL_DB_USER=root

MY_SQL_DB_PASSWORD=root

MY_SQL_DB_PORT=3306

MY_SQL_DB_DATABASE=music

MY_SQL_DB_CONNECTION_LIMIT=10

#Server Settings

PORT=5000

NODE_ENV=development

GREETING=Hello from the environment file. Be kind to the environment!

- b. After adding and configuring dotenv, and creating the .env file, test it with console.log. Print several environment variables to the console and make sure they are correct. Later, you'll get odd bugs if the .env file is not correctly read:

```
console.log(process.env.GREETING);
```

Summary of Work

We have built a simple Node.JS web server with a hand-built application structure. Later in the course when you develop Angular and React applications, you will start those applications with automatic tools. You will find that those tools generate Node.JS applications with an infrastructure of tools and source code that is similar to our hand-built application. This will help you understand how Node.JS applications

work. Node.JS applications can be as diverse as Java applications, but they all share the bones of "package.json" and its associated build, development, and deployment tools. Next, we will give our application a robust structure.

Part 2: Application Structure

The organization that develops Express describes Express as "fast, unopinionated, minimalist web framework for Node.js" (<https://expressjs.com/>). Our code so far is an example of this statement. Six lines of code in a single file are enough to get an Express web application running. However, to support a complex application, we need a professional, robust application architecture. We will develop this in the next sections.

Model-View-Controller (MVC)

In MVC, the controller responds to a user request coming from the view by manipulating the model. Think of the controller as a manager who does not know how to do any job but knows how to organize all the workers to get the job done. The model is an abstraction for the application's data. That data may have any ultimate source. The view is the end user's view and interaction with the application. The view can be a desktop application's GUI or in a browser. The view can also be another computer, communicating to an MVC server as an API consumer. That is the type of application we will be developing: a Node.JS API web server following MVC architecture.

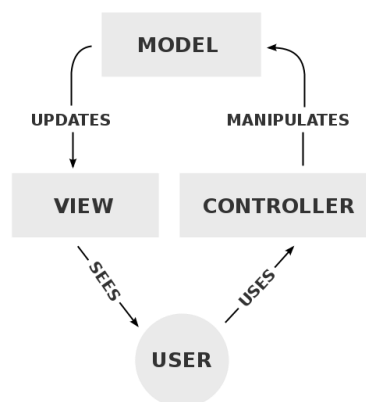


Figure 3 *MVC Process* ([RegisFrey, 2010](#))

The Router

Since we are developing a web server, an additional component not described in MVC is needed: the router. The router is in front of the controller. It checks the validity of the user request and either invokes the correct controller or rejects the request. Your web application will have both multiple controllers and multiple routers, each responding to a unique user request. Think of each unique request duplicating Figure 3 above.

Consider code from our current app.ts.

```

3  const app = express();
4  const port = 3000;
5
6  app.get('/', (req: Request, res: Response) => {
7    res.send('Hello World from TypeScript!!');
8  });
9

```

Figure 4 *Current Implementation*

The method called 'app.get' on line 6 combines a router, a controller, and a view. First, app.get defines a response to a read (get) request. When you are viewing a website, it is the result of several read requests (or 'get' requests) made by the browser to the website. That is why we can use a browser to test our simple API.. The URL `http://localhost:3000` directs the get request to the root address of a server listening on port 3000. The first parameter to app.get is '/'. This parameter is the route the request needs to follow, the root in this case. The second parameter is an arrow function that defines the controller. In this case, the controller logic is simple, responding with a "Hello, World" greeting. The response string, then, is the view.

As we develop the application, complex code is going to enter the picture and we will find the current simple structure inadequate for the task. The MusicAPI is going to support three resources, tracks, albums, and artists. Tracks will not have a direct API entry. We are going to structure our application around these resources. This gives our application a structure that will support a complex server API.

Defining the Routers and Controllers

Under the src folder, create two additional folders, 'albums' and 'artists'. For reference, at the end of this activity you will have a source tree that looks like this:

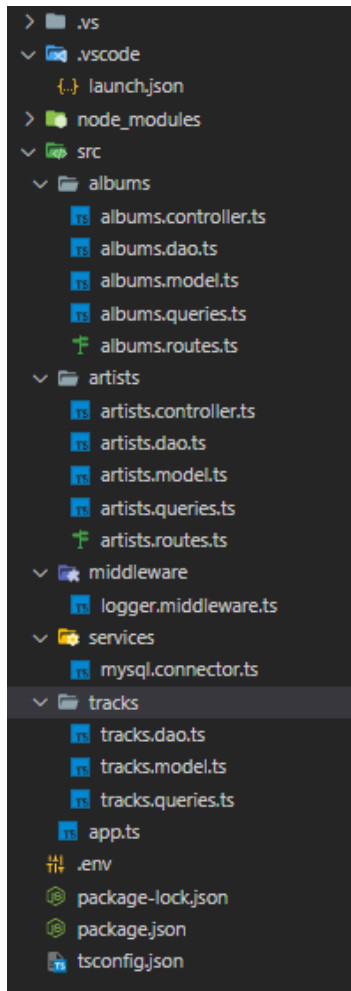


Figure 5 *Complete Source Tree*

Another useful reference is a complete list of the dependencies this project will use:

```

14  "dependencies": {
15    "cors": "^2.8.5",
16    "dotenv": "^16.0.1",
17    "express": "^4.18.1",
18    "helmet": "^5.1.0",
19    "mysql": "^2.18.1",
20    "uuid": "^8.3.2"
21  },

```

Figure 6 *All Dependencies*

Make sure your dependencies in package.json match this list. This is done using npm install.

The complete list of developer dependencies:

```

22 "devDependencies": {
23   "@types/cors": "^2.8.12",
24   "@types/dotenv": "^8.2.0",
25   "@types/express": "^4.17.13",
26   "@types/mysql": "^2.15.21",
27   "@types/uuid": "^8.3.4",
28   "nodemon": "^2.0.16",
29   "ts-node": "^10.7.0",
30   "typescript": "^4.6.4"
31 }

```

Figure 7 All Developer Dependencies

Make sure your dependencies in package.json match this list. This is done using npm install with the "—save-dev" option.

In the 'albums' folder, create the file 'albums.routes.ts' with this content':

```

src > albums > albums.routes.ts > default
1 import { Router } from 'express';
2 import { getAlbums } from './albums.controller';
3
4 const router = Router();
5 router
6   .route('/albums')
7   .get(getAlbums);
8
9 export default router;

```

Figure 8 The Albums Route

Also in the 'albums' folder, create the file 'albums.controller.ts' with this content':

```

1 import { Request, Response } from 'express';
2
3 const ALBUMS = [
4   { id: 1, name: 'Please Please Me (1963)', band: 'The Beatles' },
5   { id: 2, name: 'With The Beatles (1963)', band: 'The Beatles' },
6   { id: 3, name: 'A Hard Day\'s Night (1964)', band: 'The Beatles' },
7   { id: 4, name: 'Beatles For Sale (1964)', band: 'The Beatles' },
8   { id: 5, name: 'Help! (1965)', band: 'The Beatles' },
9   { id: 6, name: 'Rubber Soul (1965)', band: 'The Beatles' },
10  { id: 7, name: 'Sgt Pepper\'s Lonely Hearts Club Band (1967)', band: 'The Beatles' },
11  { id: 8, name: 'Most of these records are not very good', band: 'The Beatles' },
12 ];
13
14 export const getAlbums = (req: Request, res: Response) => {
15   res.send(ALBUMS);
16 };

```

Figure 9 The Albums Controller

We see in these files a router and a controller for the albums resource. Again, the router defines application routes (user requests for resources) and the controller shapes the response to those requests.

Next, in the artists folder, create 'artists.routes.ts' and 'artists.controllers.ts' with the following contents:

```
src > artists > artists.routes.ts > default
1  import { Request, Response, Router } from 'express';
2  import { getArtists } from './artists.controller';
3
4
5  const router = Router();
6  router
7    .route('/artists')
8    .get(getArtists);
9
10 export default router;
```

Figure 10 *The Artists Route*

```
src > artists > artists.controllers.ts > ARTISTS > name
1  import { Request, Response } from 'express';
2
3  const ARTISTS = [
4    { id: 1, name: 'The Beatles' },
5    { id: 2, name: 'The Who' },
6    { id: 3, name: 'Abba' },
7  ];
8
9  export const getArtists = (req: Request, res: Response) => {
10    res.send(ARTISTS);
11  };
```

Figure 11 *The Artists Controller*

To create a single set of route definitions, we need a way to unify the albums and artists routers. Notice that both albums and artists export routers named 'router.' In many programming languages, that would present a naming conflict. However, in JavaScript, modules can be assigned local names on import. This allows us to import individual routers with different, local, names. Modify 'app.ts' with the following content:

```
src > app.ts > ...
1  import express from 'express';
2  import albumsRouter from './albums/albums.routes';
3  import artistsRouter from './artists/artists.routes';
4
5  const app = express();
6  const port = 3000;
7
8  app.use('/', [albumsRouter, artistsRouter]);
9
10 app.listen(port, () => {
11   console.log(`Example app listening at http://localhost:${port}`)
12 });
13
14
```

Figure 12 *Updated 'app.ts'*

On lines 2 and 3, the albums and artists routers are imported and given local names 'albumsRouter' and 'artistsRouter'. These routes are passed via an array to the express application on line 8. With this mechanism, we can build an application with complex routing from simple, resource focused, route components.

Test the routes in a browser:

`http://localhost:3000/artists`

`http://localhost:3000/albums`

Middleware

Middleware is the heart of an Express application. In fact, our application becomes an Express application by registering our routers with 'app.use':

```
app.use('/', [albumsRouter, artistsRouter]);
```

Routers are middleware. Middleware is software inserted between the request and response:

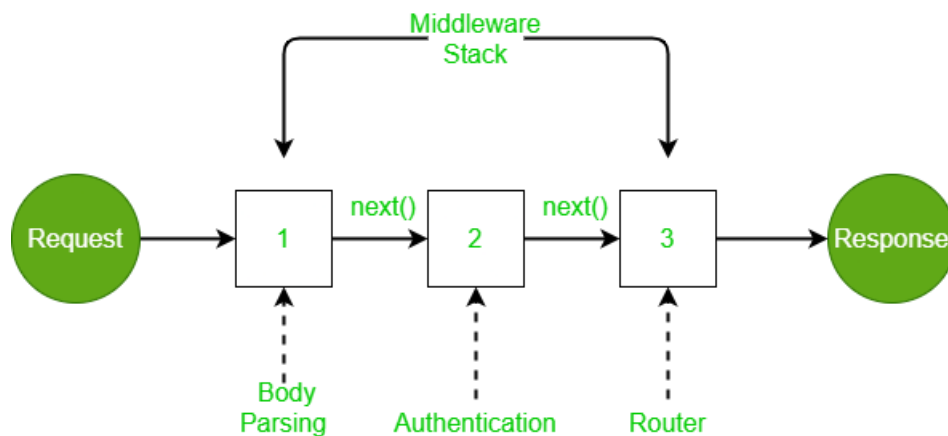


Figure 13 *Express Middleware Architecture* ([_mukul_, Geeks for Geeks](#))

In the next section, we will create a middleware logging function so you can better understand how middleware works.

A Middleware Logger

To create Express middleware, write a function that takes three parameters like this:

```
function logger(req: Request, res: Response, next: NextFunction)
```

The request and response parameters give our middleware function full visibility to the current request and response. The 'next' function calls the next function in the middleware chain. With this mechanism, middleware can stop a request from going forward by simply not calling 'next'. Our logging middleware will log all requests made to our application and always call 'next'. Create a folder called 'middleware' and put the contents in a file called 'logger.middleware.ts'. Here is the complete code:

```

1 import { Request, Response, NextFunction } from 'express';
2 import { v4 as uuidv4 } from 'uuid';
3 // import chalk from 'chalk';
4
5 const getProcessingTimeInMS = (time: [number, number]): string => {
6   return `${(time[0] * 1000 + time[1] / 1e6).toFixed(2)}ms`
7 }
8
9 /*
10  * add logs for an API endpoint using the following pattern
11  * [id][timestamp] method:url START processing
12  * [id][timestamp] method:url response.statusCode END processing
13  *
14  * @param req Express.Request
15  * @param res Express.Response
16  * @param next Express.NextFunction
17  */

```

Figure 14 Imports of 'logger.middleware.ts'

```

18 export default function logger(req: Request, res: Response, next: NextFunction) {
19   // generate unique identifier
20   const id = uuidv4();
21
22   // get timestamp
23   const now = new Date();
24   const timestamp = [now.getFullYear(), '-', now.getMonth() + 1, '-', now.getDate(), ' ', now.getHours(),
25     ':', now.getMinutes(), ':', now.getSeconds()
26   ].join('');
27
28   // get api endpoint
29   const { method, url } = req;
30
31   // log start of the execution process
32   const start = process.hrtime();
33   const startText = `START:${getProcessingTimeInMS(start)}`;
34   const idText = `[${id}]`;
35   const timeStampText = `[${timestamp}]`;
36
37   // all components are ready, show the entry
38   console.log(`${idText}${timeStampText} ${method}:${url} ${startText}`);
39
40   // trigger once a response is sent to the client
41   res.once('finish', () => {
42     // log end of the execution process
43     const end = process.hrtime(start);
44     const endText = `END:${getProcessingTimeInMS(end)}`;
45     console.log(`${idText}${timeStampText} ${method}:${url} ${res.statusCode} ${endText}`);
46   });
47
48   // execute next middleware/event handler
49   next();
50 };
51

```

Figure 15 Body of logger

This logging code has interesting features. Examine the callback registered on line 18. The callback method is invoked when the controller responds to a request. It gives us the ability to track the response time of our API.

Next, we register the middleware with Express. Modify 'app.ts' to include the following:

```
7 import logger from './middleware/logger.middleware';
```

Figure 16 *Import Logger*

```
31 if (process.env.NODE_ENV == 'development') {  
32   // add logger middleware  
33   app.use(logger);  
34   console.log(process.env.GREETING + ' in dev mode')  
35 }  
36
```

Figure 17 *Use Logger*

Next, we will add two middleware pieces that parses JSON or URL encoded request bodies and make the content ready for the application to use:

```
19 // Parse JSON bodies  
20 app.use(express.json());  
21 // Parse URL-encoded bodies  
22 app.use(express.urlencoded({ extended: true }));  
23
```

Figure 18 *Parse Content*

Next, install and add cors:

```
6 import cors from 'cors';
```

Figure 19 *Import Cors*

```
22 // enable all CORS request  
23 // needs to be installed:  
24 // npm install cors  
25 app.use(cors());
```

Figure 20 *Use Cors*

CORS or "Cross-Origin Resource Sharing" refers to the situations when a frontend running in a browser has JavaScript code that communicates with a backend, and the backend is in a different "origin" than the front end. As we've used cors() here, all cross-origin requests will be allowed by our server. This will save us from the cors error when we start developing client applications in Angular and React. These applications will communicate with the server in a cross-origin way.

Next, install and add helmet:

```
5 import helmet from 'helmet';
```

Figure 21 *Import Helmet*

```

27 // adding set of security middleware
28 // needs to be installed:
29 // npm install helmet
30 app.use(helmet());

```

Figure 22 *Use Helmet*

The middleware helmet sets security-related HTTP response headers.

Part 3: Connecting to the MySQL Database

We are now ready to replace our application's hard-coded data with an external database using MySQL. You will be provided the script necessary to create the database schema and load it with data. Two tables are created in the 'music' database: 'album' and 'track.' Familiarize yourself with these tables. You will find the creation script for this database in the "CST-391 Activity1 Music DB Current Version" zip file. The ER diagram:

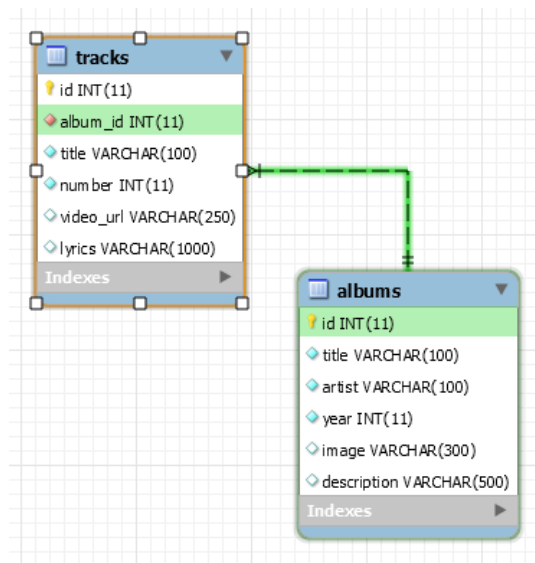


Figure 23 *ER Diagram*

MySQL Dependencies

To connect with MySQL, and exchange information with the MySQL server, we need to install a NodeJS MySQL driver. This driver provides a protocol for an application to communicate with a database. We need to install two dependencies to connect with a MySQL database:

```
npm install mysql
```

```
npm install @types/mysql --save-dev
```

The first dependency is the MySQL JavaScript driver. The second dependency adds TypeScript datatypes to the driver.

MySQL Configuration

We are going to use environment variables to provide specific MySQL connection information. We have added that in an earlier step. Verify that your MySQL connection information is correct for your setup and that you are reading those environment variables correctly.

MySQL Connection Pool

To make a request of the MySQL server, we first need to connect to the server. Once connected, we can make a query request of the server. A connection to any type of database server is an expensive transaction. The database server needs to authenticate the user and set up the communication protocol. In some cases, the connection itself can take more time than the actual query. Because of this, we use a connection pool. The connection pool saves (pooling) working connections and passes them out as requests are made. When a request is completed, the connection is released, but not closed, and it reenters the pool as an available connection. Our server is multi-user, so we request a pool allowing up to 10 simultaneous requests.

```
1 import { createPool, Pool } from 'mysql';
2 let pool: Pool | null = null;
3
4 const initializeMySQLConnector = () => {
5   try {
6
7     pool = createPool({
8       connectionLimit:
9         parseInt(process.env.MY_SQL_DB_CONNECTION_LIMIT !== undefined ? process.env.MY_SQL_DB_CONNECTION_LIMIT : ""),
10      port:
11        parseInt(process.env.MY_SQL_DB_PORT !== undefined ? process.env.MY_SQL_DB_PORT : ""),
12      host: process.env.MY_SQL_DB_HOST,
13      user: process.env.MY_SQL_DB_USER,
14      password: process.env.MY_SQL_DB_PASSWORD,
15      database: process.env.MY_SQL_DB_DATABASE,
16    });
17
18    console.debug('MySQL Adapter Pool generated successfully');
19    console.log('process.env.DB_DATABASE', process.env.MY_SQL_DB_DATABASE);
20
21    pool.getConnection((err, connection) => {
22      if (err) {
23        console.log('error mysql failed to connect');
24        throw new Error('not able to connect to database');
25      }
26      else {
27        console.log('connection made');
28        connection.release();
29      }
30    })
31  } catch (error) {
32    console.error('[mysql.connector][initializeMySQLConnector][Error]: ', error);
33    throw new Error('failed to initialized pool');
34  }
35 };
36
```

Figure 24 Create Connection Pool

Create a folder under 'src' called 'services'. In that folder create a file called 'mysql.connector.ts' with contents:

Let's unpack this code. On line 2 see the declaration:

```
let pool: Pool | null = null;
```

This gives pool two possible data types: Pool and null. In JavaScript, 'null' is both a value and a datatype. TypeScript will not allow a value to be set to null unless it has a null type. Since we want pool to be something other than 'null' we give it two datatypes: Pool | null. 'pool' is initially null and set to a Pool object on line 7. You can see here the use of the environment variables we set up in the earlier section.

The connection is tested on line 21. This preflight is probably not needed, but it provides for clear error reporting.

The 'Execute' Method

The next method we will add to 'mysql.connector.ts' is 'execute':

```
35 export const execute = <T>(query: string, params: string[] | Object): Promise<T> => {
36   try {
37     if (!pool) {
38       initializeMySQLConnector();
39     }
40
41     return new Promise<T>((resolve, reject) => {
42       pool!.query(query, params, (error, results) => {
43         if (error) reject(error);
44         else resolve(results);
45       });
46     });
47   } catch (error) {
48     console.error('[mysql.connector][execute][Error]: ', error);
49     throw new Error('failed to execute MySQL query');
50   }
51 }
52 }
```

Figure 25 Execute

The method declaration on line 35 makes 'execute' a generic method, able to return any type <T>. All SQL requests will be executed with this method at the core. In 'params', we see another example of a TypeScript declaration allowing a string array or an Object as a parameter. The method 'execute' returns a Promise. According to MDN Web Docs, "The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value." ([Promise](#), pg. 1)

What that means here is that 'execute' will return to the caller immediately with a 'Promise' object. Meanwhile, the actual operation, the SQL query, has not completed yet. The promise means "this will be finished, eventually." This changes a blocking operation into a non-blocking operation. When the query successfully completes the anonymous callback function on line 42 is invoked with 'results' set but 'error' undefined. The Promise then invokes 'resolve' with the results object. On failure, 'error' is defined, and the Promise invokes 'reject' with the error object. Either way, via reject or resolve, the caller has the response it needs.

We will see the use of the Promise from the caller's point of view later in this document when we revisit the controller methods.

Router Revisited

As this lab evolved, some changes were made in the router code. These changes use create, read, update, and delete (CRUD) as a naming convention for the internals of the application. For example, instead of 'getAlbums', 'readAlbums' becomes the controller method. Changes were also made in the way controller methods are imported. The changes:

```

1  import { Router } from 'express';
2  import * as AlbumsController from './albums.controller';
3
4  const router = Router();
5  router.
6    route('/albums').
7    get(AlbumsController.readAlbums);
8
9  export default router;

```

Figure 26 Updated Albums Router

The wild card import on line 2 allows us to give a class-like name to the methods exported by the controller. Notice the use on line 7, which shows the new naming convention and the use of AlbumsController defined on line 2. Also changed, the artist router:

```

1  import { Router } from 'express';
2  import * as ArtistsController from './artists.controller';
3
4  const router = Router();
5  router.
6    route('/artists').
7    get(ArtistsController.readArtists);
8
9  export default router;

```

Figure 27 Updated Artist Router

From the router, the next component we will modify is the controller.

The Controller Learns to Communicate with MySQL

The MusicAPI is going to support several request types. You can find the Postman-generated documentation in "MusicAPI-391 Topic 1," at <https://documenter.getpostman.com/view/8284265/UVeDtnei>. Please examine this documentation, it shows the complete API that we are working toward.

First we will look at the 'get' request, which supports two API methods:

- All albums
 - `http://localhost:5000/albums`
- Specific album matching album id
 - `http://localhost:5000/albums?albumId=7`
 - This uses a query string. A query string begins with a '?' and is a key-value pair. Additional key-value pairs can be added by separating the pairs with '&'

Modify the 'albums.controller.ts' with the following code. You can delete the methods that used hard-coded data:

```

1 import { Request, RequestHandler, Response } from 'express';
2 import { Album } from './albums.model';
3 import { Track } from '../tracks/tracks.model';
4 import * as AlbumDao from './albums.dao';
5 import * as TracksDao from '../tracks/tracks.dao';
6 import { OkPacket } from 'mysql';
7
8 export const readAlbums: RequestHandler = async (req: Request, res: Response) => {
9   try {
10     let albums;
11     let albumId = parseInt(req.query.albumId as string);
12
13     console.log('albumId', albumId);
14     if (Number.isNaN(albumId)) {
15       albums = await AlbumDao.readAlbums();
16     } else {
17       albums = await AlbumDao.readAlbumsByAlbumId(albumId);
18     }
19     await readTracks(albums, res);
20
21     res.status(200).json(
22       albums
23     );
24   } catch (error) {
25     console.error('[albums.controller][readAlbums][Error] ', error);
26     res.status(500).json({
27       message: 'There was an error when fetching albums'
28     });
29   }
30 };

```

Figure 28 Albums Controller Method (ACM)

Understanding the ACM

We will begin with a section covering the imports on lines 1 to 6. Following the imports, the body of the code starting on line 8 will be discussed in "[The Body of the Albums Controller Method \(ACM\)](#)" section below.

Data Models

On line 2 of Albums Controller Method (ACM), Album is imported. Create a file called 'albums.model.ts' in the same folder as 'albums.ts'. It is an interface that will be turned into objects for you from the server's stream of JSON data. The interface's fields should match the field names expected from the database.

In the 'albums' folder:

```
1 import { Track } from '../tracks/tracks.model';
2
3 export interface Album {
4     albumId: number,
5     title: string,
6     artist: string,
7     description: string,
8     year: string,
9     image: string,
10    tracks: Track[]
11 }
```

Figure 29 *albums.model.ts*

In the 'tracks' folder:

```
1 export interface Track {
2     trackId: number;
3     title: string;
4     number: number;
5     video: string;
6     lyrics: string;
7 }
8
```

Figure 30 *tracks.model.ts*

In the 'artists' folder:

```
1 export interface Artist {
2     artist: string;
3 }
```

Figure 31 *artists.model.ts*

A Note on Creating and Reading File Paths

In a path a single period '.' indicates the current directory. The import on line 2 imports a module located in same directory as the current module. They are sibling files in the same directory. Double periods '..' indicate the parent directory. The import on line 3 starts with the current directory, jumps to the parent directory, and then to the 'tracks' folder. If you enter the module name, VS Code will often complete the import path for you.

Data Access Objects (DAO)

The DAO methods are very short; they select the correct query (listed in the next section) and set up the parameters for that query. The heavy lifting is done by the code listed in the 'execute' method' section given earlier in this document. Since 'execute' returns a promise, the DAO methods are all marked as

'async.' Remember this means that the method will return before the job is complete, the Promise providing the communication bridge.

We will list the entire DAO files here, create files in matching folders, 'albums.dao.ts' in the 'albums' folder, and so forth. Refer to Figure 5 for clarification, if needed.

```
1  import { OkPacket } from 'mysql';
2  import { execute } from '../services/mysql.connector';
3  import { Album } from './albums.model';
4  import { albumQueries } from './albums.queries';
5
6  export const readAlbums = async () => {
7    return execute<Album[]>(albumQueries.readAlbums, []);
8  };
9
10 export const readAlbumsByArtist = async (artistName: string) => {
11   return execute<Album[]>(albumQueries.readAlbumsByArtist, [artistName]);
12 };
13
14 export const readAlbumsByArtistSearch = async (search: string) => {
15   console.log('search param', search);
16   return execute<Album[]>(albumQueries.readAlbumsByArtistSearch, [search]);
17 };
18
19 export const readAlbumsByDescriptionSearch = async (search: string) => {
20   console.log('search param', search);
21   return execute<Album[]>(albumQueries.readAlbumsByDescriptionSearch, [search]);
22 };
23
24 export const readAlbumsByAlbumId = async (albumId: number) => {
25   return execute<Album[]>(albumQueries.readAlbumsByAlbumId, [albumId]);
26 };
27
28 export const createAlbum = async (album: Album) => {
29   return execute<OkPacket>(albumQueries.createAlbum,
30     [album.title, album.artist, album.description, album.year, album.image]);
31 };
32
33 export const updateAlbum = async (album: Album) => {
34   return execute<OkPacket>(albumQueries.updateAlbum,
35     [album.title, album.artist, album.year, album.image, album.description, album.albumId]);
36 };
37
38 export const deleteAlbum = async (albumId: number) => {
39   return execute<OkPacket>(albumQueries.deleteAlbum, [albumId]);
40 };
41
```

Figure 32 *albums.dao.ts*

```
1  import { execute } from '../services/mysql.connector';
2  import { Artist } from './artists.model';
3  import { artistQueries } from './artists.queries';
4
5  export const readArtists = async () => {
6    return execute<Artist[]>(artistQueries.readArtists, []);
7  };

```

Figure 33 *artists.dao.ts*

```
1 import { execute } from '../services/mysql.connector';
2 import { Track } from './tracks.model';
3 import { trackQueries } from './tracks.queries';
4
5 export const readTracks = async (albumId: number) => {
6   return execute<Track[]>(trackQueries.readTracks, [albumId]);
7 };
8
9 export const createTrack = async (track: Track, index: number, albumId: number) => {
10   return execute<Track[]>(trackQueries.createTrack,
11     [albumId, track.title, track.number, track.video, track.lyrics]);
12 };
13
14 export const updateTrack = async (track: Track) => {
15   return execute<Track[]>(trackQueries.updateTrack,
16     [track.title, track.number, track.video, track.lyrics, track.trackId]);
17 };
```

Figure 34 *tracks.dao.ts*

Queries

The DAO code will make a lot more sense after you see its close companion, the code in 'query' files. Here is the query code for albums, from the file 'albums.queries.ts':

```
1 export const albumQueries = {
2   readAlbums: `
3     SELECT
4       id as albumId, title AS title, artist AS artist,
5       description AS description, year AS year, image AS image
6     FROM music.albums
7   `,
8   readAlbumsByArtist: `
9     SELECT
10      id as albumId, title AS title, artist AS artist,
11      description AS description, year AS year, image AS image
12    FROM music.albums
13    WHERE music.albums.artist = ?
14  `,
15  readAlbumsByArtistSearch: `
16    SELECT
17      id as albumId, title AS title, artist AS artist,
18      description AS description, year AS year, image AS image
19    FROM music.albums
20    WHERE music.albums.artist LIKE ?
21  `,
22  readAlbumsByDescriptionSearch: `
23    SELECT
24      id as albumId, title AS title, artist AS artist,
25      description AS description, year AS year, image AS image
26    FROM music.albums
27    WHERE music.albums.description LIKE ?
28  `,
29  readAlbumsByAlbumId: `
30    SELECT
31      id as albumId, title AS title, artist AS artist,
32      description AS description, year AS year, image AS image
33    FROM music.albums
34    WHERE music.albums.id = ?
35  `,
36  createAlbum: `
37    INSERT INTO ALBUMS(title, artist, description, year, image) VALUES(?,?,?,?,:)
38  `,
39  updateAlbum: `
40    UPDATE music.albums
41    SET title = ?, artist = ?, year = ?, image = ?, description = ?
42    WHERE id = ?
43  `,
44  deleteAlbum: `
45    DELETE FROM music.albums
46    WHERE id = ?
47  `,
48 }
```

Figure 35 *albums.queries.ts*

As you can see, the query object is a set of key-value pairs. Notice that the value is a JavaScript template literal, a special type of string that is enclosed in backticks (`). You use the key and the value is passed to the 'execute' function of the DAO. The second parameter to 'execute' is an array of parameters. When a question mark appears in the query, the ? is replaced by the parameter. When there is more than one

parameter, it is important to carefully match the parameter order with the order of the question marks that appear in the query. Look 'updateAlbum' and 'createAlbum' for examples of how multiple parameters are used.

Here are the other query code files:

```
1 export const artistQueries = {
2   readArtists: `
3     SELECT
4       DISTINCT artist as artist
5     FROM music.albums
6   `
7 }
```

Figure 36 *artists.queries.ts*

```
1 export const trackQueries = {
2   createTrack: `
3     INSERT INTO tracks (album_id, title, number, video_url) VALUES(?,?,?,?)
4   `,
5   readTracks: `
6     SELECT title AS title, video_url AS video, lyrics AS lyrics
7     FROM music.tracks
8     WHERE album_id = ?
9   `,
10  updateTrack: `
11    UPDATE music.tracks
12    SET title = ?, number = ?, video_url = ?, lyrics = ?
13    WHERE id = ?;
14  `
15 }
```

Figure 37 *tracks.queries.ts*

The Body of the Albums Controller Method (ACM)

If you look back at the earlier code listing for the [Albums Controller Method \(ACM\)](#), you'll see we've covered up to line 5 (part of the library imports). On line 6, OkPacket is imported from the MySQL library. OkPacket is the type that is returned when status information is returned by MySQL rather than a data set as with the SELECT query. This is the mechanism for returning the automatically (by MySQL) assigned primary key for new records.

Now you're ready to understand the body of the controller method 'readAlbums'; on line 8, you see the 'async' keyword. This can only be used on methods that use 'await', which we see on lines 15, 17, and 19. In turn, 'await' can only be used on methods that return Promises. So 'async' methods return to the caller as soon as they hit 'await'. This makes them non-blocking from the caller's perspective. Meanwhile, the 'await' call is waiting for the Promise to either 'resolve' or 'reject'. You can see this in the 'execute' method in 'mysql.connector.ts'.

Think about why this asynchronous mechanism is used. The Express web server we are building will support multiple simultaneous clients. If you remove the Promise and with it the use of 'async' and 'await', an individual request of the server will still work. However, the server itself will be completely blocked while that request is served, unable to accept any other client connection. To those clients, the server will appear to be offline and unavailable. By using a Promise, the controller method returns almost

immediately, which allows the next client to get served immediately. No client gets its data back immediately, but it can connect to begin its request.

The 'readAlbums' method supports URL query strings. Query string work like optional parameters in a URL. The query string begins with a '?' and then is a set of key-value pairs separated by '&'. Since query strings are optional, the router cannot use the query string to differentiate between different requests. An example of a query string taken from [Query String](#), pg. 1:

`https://example.com/path/to/page?name=ferret&color=purple`

In this example, "name=ferret" is one key-value pair and "color=purple" is another. The routed path is `https://example.com/path/to/page`. The controller method typically pulls the parameters from the query string.

In the MusicAPI, we have two possibilities to retrieve albums, which illustrate the use of query strings:

<http://localhost:5000/albums>

<http://localhost:5000/albums?albumId=7>

Both of these methods will be routed to the 'readAlbums' method. On line 11, the controller is looking for the possible query parameter:

```
let albumId = parseInt(req.query.albumId as string);
```

If the query parameter is absent, 'albumId' will get the value 'NaN', or Not a Number. If a query parameter is set and a number, then albumId will be set to that number:

```
if (Number.isNaN(albumId)) {  
  albums = await AlbumDao.readAlbums();  
} else {  
  albums = await AlbumDao.readAlbumsByAlbumId(albumId);  
}
```

Complete Controllers

Now that we have discussed the router, controller, and DAO code, here is the complete listing of our controller code. In MusicAPI, we have two controllers, 'albums' and 'artists'. The MusicAPI has tracks as a resource, but tracks are not available directly through the API and so it doesn't have either router or controller code.

The other methods of 'albums.controller.ts':

```

32 export const readAlbumsByArtist: RequestHandler = async (req: Request, res: Response) => {
33   try {
34     const albums = await AlbumDao.readAlbumsByArtist(req.params.artist);
35
36     await readTracks(albums, res);
37
38     res.status(200).json(
39       albums
40     );
41   } catch (error) {
42     console.error('[albums.controller][readAlbums][Error] ', error);
43     res.status(500).json({
44       message: 'There was an error when fetching albums'
45     });
46   }
47 };

```

Figure 38 *albums.controller.ts*, *readAlbumsByArtist*

```

49 export const readAlbumsByArtistSearch: RequestHandler = async (req: Request, res: Response) => {
50   try {
51     console.log('search', req.params.search);
52     const albums = await AlbumDao.readAlbumsByArtistSearch('%' + req.params.search + '%');
53
54     await readTracks(albums, res);
55
56     res.status(200).json(
57       albums
58     );
59   } catch (error) {
60     console.error('[albums.controller][readAlbums][Error] ', error);
61     res.status(500).json({
62       message: 'There was an error when fetching albums'
63     });
64   }
65 };

```

Figure 39 *albums.controller.ts*, *readAlbumsByArtistSearch*

```

67 export const readAlbumsByDescriptionSearch: RequestHandler = async (req: Request, res: Response) => {
68   try {
69     console.log('search', req.params.search);
70     const albums = await AlbumDao.readAlbumsByDescriptionSearch('%' + req.params.search + '%');
71
72     await readTracks(albums, res);
73
74     res.status(200).json(
75       albums
76     );
77   } catch (error) {
78     console.error('[albums.controller][readAlbums][Error] ', error);
79     res.status(500).json({
80       message: 'There was an error when fetching albums'
81     });
82   }
83 };

```

Figure 40 *albums.controller.ts*, *readAlbumsByDescriptionSearch*

```

85 export const createAlbum: RequestHandler = async (req: Request, res: Response) => {
86   try {
87     const okPacket: OkPacket = await AlbumDao.createAlbum(req.body);
88
89     console.log('req.body', req.body);
90
91     console.log('album', okPacket);
92
93     req.body.tracks.forEach(async (track: Track, index: number) => {
94       try {
95         await TracksDao.createTrack(track, index, okPacket.insertId);
96       } catch (error) {
97         console.error('[albums.controller][createAlbumTracks][Error] ', error);
98         res.status(500).json({
99           message: 'There was an error when writing album tracks'
100         });
101       }
102     });
103
104     res.status(200).json(
105       okPacket
106     );
107   } catch (error) {
108     console.error('[albums.controller][createAlbum][Error] ', error);
109     res.status(500).json({
110       message: 'There was an error when writing albums'
111     });
112   }
113 };

```

Figure 41 *albums.controller.ts*, *createAlbum*


```

115 export const updateAlbum: RequestHandler = async (req: Request, res: Response) => {
116   try {
117     const okPacket: OkPacket = await AlbumDao.updateAlbum(req.body);
118
119     console.log('req.body', req.body);
120
121     console.log('album', okPacket);
122
123     req.body.tracks.forEach(async (track: Track, index: number) => {
124       try {
125         await TracksDao.updateTrack(track);
126       } catch (error) {
127         console.error('[albums.controller][updateAlbum][Error] ', error);
128         res.status(500).json({
129           message: 'There was an error when updating album tracks'
130         });
131       }
132     });
133
134     res.status(200).json(
135       okPacket
136     );
137   } catch (error) {
138     console.error('[albums.controller][updateAlbum][Error] ', error);
139     res.status(500).json({
140       message: 'There was an error when updating albums'
141     });
142   }
143 };

```

Figure 42 *albums.controller.ts, updateAlbum*

```

145 async function readTracks(albums: Album[], res: Response<any, Record<string, any>>) {
146   for (let i = 0; i < albums.length; i++) {
147     try {
148       const tracks = await TracksDao.readTracks(albums[i].albumId);
149       albums[i].tracks = tracks;
150
151     } catch (error) {
152       console.error('[albums.controller][readTracks][Error] ', error);
153       res.status(500).json({
154         message: 'There was an error when fetching album tracks'
155       });
156     }
157   }
158 }

```

Figure 43 *albums.controller.ts, readTracks (private)*

```

160 export const deleteAlbum: RequestHandler = async (req: Request, res: Response) => {
161   try {
162     let albumId = parseInt(req.params.albumId as string);
163
164     console.log('albumId', albumId);
165     if (!Number.isNaN(albumId)) {
166       const response = await AlbumDao.deleteAlbum(albumId);
167
168       res.status(200).json(
169         response
170       );
171     } else {
172       throw new Error("Integer expected for albumId");
173     }
174
175   } catch (error) {
176     console.error('[albums.controller][deleteAlbum][Error] ', error);
177     res.status(500).json({
178       message: 'There was an error when deleting albums'
179     });
180   }
181 };

```

Figure 44 *albums.controller.ts*, *deleteAlbum*

To complete the 'albums' code, here is the complete router for albums:

```

1  import { Router } from 'express';
2  import * as AlbumsController from './albums.controller';
3
4  const router = Router();
5  router.
6    route('/albums').
7      get(AlbumsController.readAlbums);
8
9  router.
10    route('/albums/:artist').
11      get(AlbumsController.readAlbumsByArtist);
12
13  router.
14    route('/albums/search/artist/:search').
15      get(AlbumsController.readAlbumsByArtistSearch);
16
17  router.
18    route('/albums/search/description/:search').
19      get(AlbumsController.readAlbumsByDescriptionSearch);
20
21  router.
22    route('/albums').
23      post(AlbumsController.createAlbum);
24
25  router.
26    route('/albums').
27      put(AlbumsController.updateAlbum);
28
29  router.
30    route('/albums/:albumId').
31      delete(AlbumsController.deleteAlbum);
32
33  export default router;

```

Figure 45 *albums.routes.ts*

The route segments beginning with a ':' indicate a variable segment. The variable segment will match any Uri string and Express will make that available to your application with a key property that matches the variable segment name. Examine the controller code that reads these values.

The Artist Controller

In the MusicAPI, the albums controller is the elaborated controller, as you have seen. In contrast, the artist controller is simple because our API will only support the GET verb. Here it is the complete artists controller:

```
1 import { Request, RequestHandler, Response } from 'express';
2 import * as ArtistDao from './artists.dao';
3
4
5 export const readArtists: RequestHandler = async (req: Request, res: Response) => {
6   try {
7     const artists = await ArtistDao.readArtists();
8
9     res.status(200).json(
10      artists
11    );
12   } catch (error) {
13     console.error('[artists.controller][ReadArtists][Error] ', error);
14     res.status(500).json({
15       message: 'There was an error when fetching artists'
16     });
17   }
18 };
```

Figure 46 *artists.controller.ts*

One reason why the artists side of the code is simple is that there is no artists data table to maintain. If you examine the *artists.query.ts* file, you will see that the query pulls the artist information from the album database.

Reviewing the MusicAPI

Code Structure

The Express library is not opinionated. In practice, that means you can structure your Express application in any way you wish. However, in practice, applications need a structure to be maintainable. The application structure presented in this activity is model-view-controller (MVC). Since it is an API, the view is the data response to requests.

RESTful API

The MusicAPI is a REST API. This is a deep topic, but the fundamentals of a REST API are not hard to understand:

- An application programmer interface (API) allows a client computer to extract and interact with information on a server computer.
- A REST API is built in conformance to the way webpages interact with web browsers. Because of this, it looks like common internet traffic to routers that are tuned to handling common traffic.

- REST does not require HTTP as a transport protocol, but this is implied since REST conforms to HTTP.
- HTTP is built on request/response with the server being a stateless machine. Each request/response should be an independent event for the server.
- REST Uris should be based on plural nouns only. These nouns are called 'resources'. The action taken is expressed in the HTTP verb.
 - It is tempting to write API methods that read and write products as
 - `https://somewebsserver/readProducts`
 - HTTP verb: GET
 - `https://somewebsserver/writeProducts`
 - HTTP verb: POST
 - Creating a URI that includes both a noun and a verb (e.g., 'readProducts') conforms to the way we are taught to write methods in a programming language. However, this does not conform to REST naming conventions. Below are the same API methods written using REST conventions.
 - `https://somewebsserver/products`
 - HTTP verb: GET
 - `https://somewebsserver/products`
 - HTTP verb: POST

Submission

Submit the following as directed by the instructor:

1. A Microsoft Word document with an activated (blue) link to a screencast demonstrating the use of the MusicAPI in Postman.
2. In the screencast itself, you will demonstrate:
 - a. All nine required API entry points; please refer to "MusicAPI-391 Topic 1," located in the topic Resources, for a complete definition.
 - b. Drill down into the details of one API method, your choice, discussing the router, the controller, and the DAO of that API method.

Appendix: Implementation Notes

Note: If you ever need to rebuild the `node_modules` directory in a project, just delete the directory and run the `npm install` command within the projects *root* directory.

Complete Code Listing for *app.ts*

As noted, Express is built on middleware. That middleware is sequentially executed by Express. As a developer, you control the order of that sequence by the order of calls to `app.use`. We developed *app.ts* in pieces. Make sure your middleware sequence is in order by comparing your *app.ts* code to the *app.ts* listing:

```
1  import express, { Request, Response } from 'express';
2  import albumsRouter from './albums/albums.routes';
3  import artistsRouter from './artists/artists.routes';
4  import helmet from 'helmet';
5  import cors from 'cors';
6  import logger from './middleware/logger.middleware';
7  import dotenv from "dotenv";
8
9  dotenv.config();
10
11 const app = express();
12 const port = process.env.PORT;
13
14 // enable all CORS request
15 app.use(cors());
16
17 // Parse JSON bodies
18 app.use(express.json());
19 // Parse URL-encoded bodies
20 app.use(express.urlencoded({ extended: true }));
21
22 // adding set of security middleware
23 app.use(helmet());
24
25 console.log(process.env.MY_SQL_DB_HOST);
```

Figure 47 Part 1: *app.ts*

```

26
27 //MySQLConnector.initializeMySQLConnector();
28
29 if (process.env.NODE_ENV == 'development') {
30     // add logger middleware
31     app.use(logger);
32     console.log(process.env.GREETING + ' in dev mode')
33 }
34
35 // Application routes
36 // root route
37 app.get('/', (req: Request, res: Response) => {
38     res.send('<h1>Welcome to the Music API<h1/>');
39 });
40 // adding router middleware
41 app.use('/', [albumsRouter , artistsRouter] );
42
43
44 app.listen(port, () => {
45     console.log(`Example app listening at http://localhost:${port}`)
46 });
47

```

Figure 48 Part 2: *app.ts*

Supplemental Resources

Supplemental Resources

Reference the following resources as necessary when completing the activity:

- **TypeScript Home Page:** <https://www.typescriptlang.org>
- **TypeScript Tutorial:** <https://www.tutorialspoint.com/typescript/index.htm>
- **ES6 Tutorial:** <https://www.tutorialspoint.com/es6/index.htm>
- **ES6 Tutorial (Understanding Callback):**
https://www.tutorialspoint.com/es6/es6_promises.htm
- **Asynchronous JavaScript:** <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous>
- **JavaScript Promises: An Introduction:**
<https://developers.google.com/web/fundamentals/primers/promises>
- **Promises in NodeJS:** Go to the NodeJS documentation home page for the version of NodeJS that was installed for this course. Read the documentation on the `util.promisify()` method in the Utility Library.

Supplemental Tutorials

Reference the following tutorials as necessary when completing the activity:

- **TypeScript Express Tutorial #1:** <https://wanago.io/2018/12/03/typescript-express-tutorial-routing-controllers-middleware/>
- **Learn How to Use TypeScript With Node.js and Express.js:** <https://www.becomebetterprogrammer.com/learn-how-to-use-typescript-with-node-js-and-express-js/>
- **Node.js Logging Tutorial:** <https://stackify.com/node-js-logging/>
- **How to Use MySQL in Node.js and Express.js with TypeScript:** <https://www.becomebetterprogrammer.com/mysql-nodejs-expressjs-typescript/>
- **Your Guide to Building a NodeJS, TypeScript Rest API with MySQL:** <https://livecodestream.dev/post/your-guide-to-building-a-nodejs-typescript-rest-api-with-mysql/>