CST-391 Activity 1 Guide

Contents

[Part 1: Refining the Build 2](#_Toc127347428)

[Overview 2](#_Toc127347429)

[Scripts in package.json 2](#_Toc127347430)

[Summary of Work 3](#_Toc127347431)

[Part 2: Application Structure 4](#_Toc127347432)

[Model-View-Controller (MVC) 4](#_Toc127347433)

[The Router 4](#_Toc127347434)

[Middleware 9](#_Toc127347435)

[Part 3: Connecting to the MySQL Database 12](#_Toc127347436)

[MySQL Dependencies 12](#_Toc127347437)

[MySQL Configuration 12](#_Toc127347438)

[MySQL Connection Pool 13](#_Toc127347439)

[Router Revisited 14](#_Toc127347440)

[The Controller Learns to Communicate with MySQL 15](#_Toc127347441)

[Reviewing the MusicAPI 27](#_Toc127347442)

[Code Structure 27](#_Toc127347443)

[RESTful API 27](#_Toc127347444)

[Submission 28](#_Toc127347445)

[Appendix: Implementation Notes 29](#_Toc127347446)

[Complete Code Listing for *app.ts* 29](#_Toc127347447)

[Supplemental Resources 30](#_Toc127347448)

## 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",

1. 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"

}

1. 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

1. 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.
2. 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

1. 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:

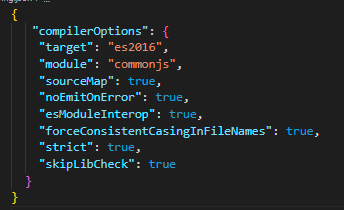


Figure 1 *tsconfig.json*

1. 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:

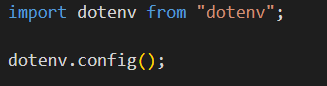


Figure 2 *Preparing Environment Variables*

dotenv

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

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

* 1. 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!

* 1. 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](https://commons.wikimedia.org/wiki/File:MVC-Process.svg))

### 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.

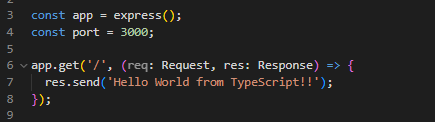


Figure 4 *Current Implementation*

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:

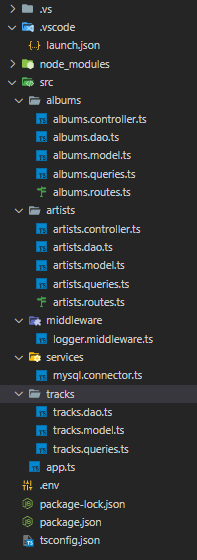


Figure 5 *Complete Source Tree*

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

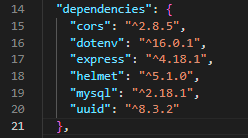


Figure 6 *All Dependencies*

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

The complete list of developer dependencies:

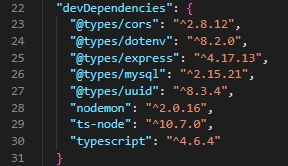


Figure 7 *All Developer Dependencies*

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':

Text

Description automatically generated

Figure 8 *The Albums Route*

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

Text

Description automatically generated

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:

Text

Description automatically generated

Figure 10 *The Artists Route*

A screenshot of a computer

Description automatically generated with medium confidence

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:

Text

Description automatically generated

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:

Graphical user interface, application

Description automatically generated

Figure 13 *Express Middleware Architecture* ([\_mukul\_, Geeks for Geeks](https://www.geeksforgeeks.org/middleware-in-express-js/))

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:

Text

Description automatically generated

Figure 14 *Imports of 'logger.middleware.ts'*

Figure 15 *Text

Description automatically generatedBody 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:



Figure 16 *Import Logger*

Text

Description automatically generated

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:

Text

Description automatically generated

Figure 18 *Parse Content*

Next, install and add cors:



Figure 19 *Import Cors*

Text

Description automatically generated

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:



Figure 21 *Import Helmet*

Text

Description automatically generated

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:

Graphical user interface, diagram

Description automatically generated

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.

Text

Description automatically generated

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 prefight 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':

Text

Description automatically generated

#### 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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:

Text

Description automatically generated

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:

Text

Description automatically generated

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:

Text

Description automatically generated

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:

Text

Description automatically generated  
Figure 29 *albums.model.ts*

In the 'tracks' folder:

Text

Description automatically generated

Figure 30 *tracks.model.ts*

In the 'artists' folder:

Graphical user interface, text

Description automatically generated

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.

Text

Description automatically generated

Figure 32 *albums.dao.ts*

Text

Description automatically generated

#### Figure 33 artists.dao.ts

Text

Description automatically generated

#### 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':

Text

Description automatically generated

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:

Text

Description automatically generated

Figure 36 *artists.queries.ts*

Text

Description automatically generated

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)](#_Albums_Controller_Method), 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 'myql.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](https://en.wikipedia.org/wiki/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':

#### Text Description automatically generated

Figure 38 *albums.controller.ts, readAlbumsByArtist*

#### Text Description automatically generated

Figure 39 *albums.controller.ts, readAlbumsByArtistSearch*

#### Text Description automatically generated

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

#### Text Description automatically generated

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

#### Text Description automatically generated

#### Figure 42 albums.controller.ts, updateAlbum

#### A screenshot of a computer Description automatically generated with medium confidence

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

#### Text Description automatically generated

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

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

Text

Description automatically generated

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:

Text

Description automatically generated

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://somewebserver/readProducts
      * HTTP verb: GET
    - https://somewebserver/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://somewebserver/products
      * HTTP verb: GET
    - https://somewebserver/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:
   1. All nine required API entry points; please refer to "MusicAPI-391 Topic 1," located in the topic Resources, for a complete definition.
   2. 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:

Text

Description automatically generated

Figure 47 *Part 1: app.ts*

**

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/>