**X-Press Publishing**

In this project, you will build a complete API using Express.js and SQLite for an internal management tool for a comic book publishing company: X-Press Publishing.

If you get stuck during this project or would like to see an experienced developer work through it, click “**Get Help**“ to see a **project walkthrough video**.

**Tasks**

**54/54Complete**

Mark the tasks as complete by checking them off

**Setup Project**

**1.**

Download and unzip the skeleton code for this project [here](https://s3.amazonaws.com/codecademy-content/programs/build-apis/projects/capstone-project-1-x-press-publishing.zip).

**2.**

Navigate to the root directory using a terminal and run npm install.

**Setup Server**

**3.**

In the project’s root directory, create a file named **server.js**.

**4.**

Install and import the following packages: body-parser, cors, errorhandler, morgan, and express. When installing, ensure they get saved to the dependencies in **package.json**.

Hint

For example, to install and save body-parser, we would run npm install body-parse --save in the project in a terminal.

To import body-parser and save it to a variable we would write const bodyParser = require('body-parser')

**5.**

Create an instance of an Express app and save it to a variable.

Hint

const app = express()

**6.**

Create a variable representing the port your server will listen on. This should be able to be optionally set to process.env.PORT if that value is set, for testing purposes. Otherwise, it should default to the port number of your choice (except 8081 as this is the port the test file uses).

Hint

const PORT = process.env.PORT || 4000

**7.**

Use the body-parser JSON, errorhandler, and CORS middleware functions for all routes in the server. Additionally consider setting up morgan logging to the dev setting for easier debugging.

Hint

For example, to use the body-parser JSON middleware, you would include:

app.use(bodyParser.json())

**8.**

Start your server using the port variable you declared earlier. Add a callback function that will log out a useful message to the console once your server is running.

Hint

app.listen(PORT, () => {

console.log('Listening on port: ' + PORT);

});

**9.**

Finally, export the Express app (for use in the test file).

Hint

module.exports = app

**Create API Router**

**10.**

Since all routes in this project have paths starting at the /api subpath, we will create an API router that will prepend this path segment.

Create a directory called **api/** in the root directory of the project. Within this directory, create a file called **api.js**.

**11.**

Within **api.js**, importexpress. Then create an instance of an Express router and save it to a variable.

Hint

const apiRouter = express.Router()

**12.**

Export the router.

**13.**

In **server.js**, import the API router and mount it at all routes starting at /api.

Hint

const apiRouter = require('./api/api');

// Other server code.

app.use('/api', apiRouter);

**Create Artist Table**

**14.**

Next, we will write the code to create the Artist table. In the root directory, open **migration.js**.

Import sqlite3. It should already be installed from when you ran npm install at the beginning of the project.

**15.**

Open the **database.sqlite** file as a sqlite3 database object and save it to a variable.

Hint

const db = new sqlite3.Database('./database.sqlite');

**16.**

On the database object, run a SQLite command to create an Artist table (if it doesn’t already exist) with the following schema:

* **id** - Integer, primary key, required
* **name** - Text, required
* **date\_of\_birth** - Text, required
* **biography** - Text, required
* **is\_currently\_employed** - Integer, defaults to 1

Hint

db.run('CREATE TABLE IF NOT EXISTS `Artist` ( ' +

'`id` INTEGER NOT NULL, ' +

'`name` TEXT NOT NULL, ' +

'`date\_of\_birth` TEXT NOT NULL, ' +

'`biography` TEXT NOT NULL, ' +

'`is\_currently\_employed` INTEGER NOT NULL DEFAULT 1, ' +

'PRIMARY KEY(`id`) )');

**17.**

Once you’ve finished writing the migration. Run node migration.js in a terminal to run the migration.

Once the migration has finished running, open **database.sqlite** using DB Browser to ensure your table looks as expected. Then try inserting values into the table to ensure it works.

When you think your table is ready, run the testing suite by running npm test in a terminal to check your work for the Artist table.

**Create Artist Routes**

**18.**

Now, let’s write the CRUD operations for /api/artists.

Start by creating a file in the **api/** directory called **artists.js**. Within **artists.js**, import express, create an Express router, and export the Express router.

**19.**

In **api.js**, import the artists router and mount it at /artists.

**20.**

Back in **artists.js**, import sqlite3 and open an instance of **database.sqlite**. When loading the database, check if process.env.TEST\_DATABASE has been set, and if so load that database instead. This will allow the testing suite to check your routes without corrupting your database.

Hint

const db = new sqlite3.Database(process.env.TEST\_DATABASE || './database.sqlite')

**21.**

Add a GET handler to the router for the / path (remember this router is already mounted at /api/artists so there is no need to add all of that boilerplate).

Hint

artistsRouter.get('/', (req, res, next) => {

});

**22.**

Within the GET / handler, execute a sqlite3 query that will retrieve all entries in the Artist table where is\_currently\_employed is equal to 1. This will retrieve all currently-employed artists.

Hint

db.all('SELECT \* FROM Artist WHERE Artist.is\_currently\_employed = 1', (err, artists) => {

});

**23.**

Within the callback of this query, check if there is an error. If so, pass it along the middleware chain to be dealt with by errorhandler.

Otherwise, send a response containing a JSON body with a key of artists and a value of the retrieved artists, along with a status code of 200.

When you think your /api/artists GET route is ready, run the testing suite to check your work.

Hint

if (err) {

next(err);

} else {

res.status(200).json({artists: artists});

}

**24.**

Our GET, PUT, and DELETE route paths will have an :artistId parameter. For both, we will have to check that an artist with that ID exists and if not send a 404 response. Let’s add a router param to reduce boilerplate code.

Add a router param of artistId to the router.

Stuck? Get a hint

**25.**

Within the artist param callback, execute a SQL query which will get the artist with the given ID.

Stuck? Get a hint

**26.**

Within the callback function of the sqlite3 query, send any SQL errors down the middleware chain. Otherwise, check if an artist was retrieved. If so, attach it to the request object as artist and move to the next function in the middleware chain. If not, send a 404 response.

Stuck? Get a hint

**27.**

Now we’re ready to write our GET /api/artist/:artistId handler.

Register another GET handler at the above path segment. Our router param should already handle all of the necessary SQL and error-handling logic and attach the retrieved artist at req.artist.

Within the handler’s callback function, return a 200 response with a JSON body containing a key of artist and a value of the retrieved artist.

When you think your /api/artists/:artistId GET route is ready, run the testing suite to check your work.

Stuck? Get a hint

**28.**

Add a POST handler at the / path segment. Within the callback function of this handler, check that all required fields are present on the artist object of the request body (name, dateOfBirth, and biography). If not, send a 400 response.

Stuck? Get a hint

**29.**

Additionally check to see if is\_currently\_employed was set on the request’s artist object. If not, set it to 1. This will simplify our SQL logic in the next step.

Hint

const isCurrentlyEmployed = req.body.artist.isCurrentlyEmployed === 0 ? 0 : 1;

**30.**

Next, execute a SQL query to create a new Artist with the supplied attributes.

Hint

const sql = 'INSERT INTO Artist (name, date\_of\_birth, biography, is\_currently\_employed) VALUES ($name, $dateOfBirth, $biography, $isCurrentlyEmployed)';

const values = {

$name: name,

$dateOfBirth: dateOfBirth,

$biography: biography,

$isCurrentlyEmployed: isCurrentlyEmployed

};

db.run(sql, values, function(error) {

});

**31.**

In the callback function of this SQL query, check for errors and pass them down the middleware chain. If no errors are present, you will need to retrieve the newly-created artist from the database to return it in the response.

Using the sqlite3 lastID attribute, write a SQL query to retrieve the new artist from the database. Then send it in the response body with a 201 status code.

When you think your POST route is ready, run the testing suite to check your work.

Hint

db.get(`SELECT \* FROM Artist WHERE Artist.id = ${this.lastID}`,

(error, artist) => {

res.status(201).json({artist: artist});

});

**32.**

Add a PUT handler at /:artistId to your router. Check to ensure all required fields are present in the request body, if not send a 400 response.

**33.**

Execute a SQL statement to update the artist with the supplied artist ID to have the supplied updated attributes.

Hint

const sql = 'UPDATE Artist SET name = $name, date\_of\_birth = $dateOfBirth, biography = $biography, is\_currently\_employed = $isCurrentlyEmployed WHERE Artist.id = $artistId';

const values = {

$name: name,

$dateOfBirth: dateOfBirth,

$biography: biography,

$isCurrentlyEmployed: isCurrentlyEmployed,

$artistId: req.params.artistId

};

db.run(sql, values, (error) => {

});

**34.**

In the callback of the update statement, pass any errors down the middleware chain, if present. Otherwise, retrieve the newly-updated artist from the database and send it in the response with a 200 status code.

When you think your PUT route is ready, run the testing suite to check your work.

**35.**

Finally, let’s add our delete handler. This will be slightly different than a normal delete — instead of deleting the artist, we will mark them as unemployed.

Add a delete handler at /:artistId. Within the handler’s callback function, run a SQL query to update the artist with the supplied artist ID to have is\_currently\_employed equal to 0.

Handle any errors and send successfully-updated artists with 200 responses.

When you think your DELETE route is ready, run the testing suite to check your work.

At this point when you load the X-Press Publishing app, a list of all saved artists should load on the landing page. Clicking one of them should allow you to view, update, and delete that artist. You should additionally be able to create new artists by clicking the ‘New Artist’ button on the landing page.

**Create Series Table**

**36.**

Now, let’s move on to our next model. Open **migration.js** and write the code to create a new table called Series, if one doesn’t already exist. The table should have the following column properties:

* **id** - Integer, primary key, required
* **name** - Text, required
* **description** - Text, required

**37.**

Run **migration.js** via command line. Use DB Browser to ensure your table was properly set up - examining table properties and inserting data into the table. When you think your table is ready, run the testing suite to check your work.

**Create Series Routes**

**38.**

Let’s add our /api/series routes.

Create **series.js** in the **api/** directory. In **series.js**, create and export an Express router.

Import the router into **api.js** and mount it at /series.

**39.**

In **series.js**, open a sqlite3database to your database or the provided test database.

**40.**

Add a GET handler at / that will retrieve all existing series from the database.

All errors should be properly handled and successfully retrieved series should be returned on the series property of the response object with a 200 status code.

When you think the route is ready, run the testing suite to check your work.

**41.**

Add a router param for handling the seriesId parameter. This router param should retrieve the specified series from the database and attach it as series on the request object. If a series with that ID does not exist, a 404 response should be sent.

When you think the route is ready, run the testing suite to check your work.

**42.**

Add a GET /:seriesId route. This route should return the retrieved series on the request object with a 200 status code.

When you think the route is ready, run the testing suite to check your work.

**43.**

Add a POST / route. This route should return a 400 response if any required fields are missing (name or description). Otherwise, it should add the new series to the database and return the newly-created series with a 201 status code, handling any errors along the way.

When you think the route is ready, run the testing suite to check your work.

**44.**

Add a PUT /:seriesId route. This route should return a 400 response if any required fields are missing (name or description). Otherwise, it should add the new series to the database and return the newly-updated series with a 200 status code, handling any errors along the way.

When you think the route is ready, run the testing suite to check your work.

We will wait to write the DELETE /:seriesId route until we’ve added issues, as checking for whether or not a series has issue prior to deletion is a large part of the logic.

At this point when you load the X-Press Publishing app, a list of all saved series should load on the landing page. Clicking one of them should allow you to view and update that series. You should additionally be able to create new series by clicking the ‘New Series’ button on the landing page.

**Create Issue Table**

**45.**

Let’s move on to our last model. Open **migration.js** and write the code to create a new table called Issue, if one doesn’t already exist. The table should have the following column properties:

* **id** - Integer, primary key, required
* **name** - Text, required
* **issue\_number** - Integer, required
* **publication\_date** - Text, required
* **artist\_id** - Integer, foreign key, required
* **series\_id** - Integer, foreign key, required

Hint

A foreign key constraint looks like this: FOREIGN KEY(artist\_id) REFERENCES Artist(id)

**46.**

Run **migration.js** via command line. Use DB Browser to ensure your table was properly set up - examining table properties and inserting data into the table. When you think your table is ready, run the testing suite to check your work.

**Create Issue Routes**

**47.**

Let’s add our /api/series/:seriesId/issues routes.

Create **issues.js** in the **api/** directory. In **issues.js**, create and export an Express router.

Import the router into **series.js** and mount it at /:seriesId/issues.

Since you’ll need to access information about the specified series from the issues router, you’ll need to merge parameters. Add the approporiate paramter to your issue router instantiation to enable merging parameters.

Hint

const issuesRouter = express.Router({mergeParams: true});

**48.**

Add a GET handler at / that will retrieve all existing issues of the specified series from the database.

All errors should be properly handled and successfully retrieved issues should be returned on the issues property of the response object with a 200 status code.

When you think the route is ready, run the testing suite to check your work.

**49.**

Add a POST / route. This route should return a 400 response if any required fields are missing (name, issueNumber, publicationDate, or artistId). Additionally, the route should return a 400 response if an artist with the provided artist ID does not exist. Otherwise, it should add the new issue to the database and return the newly-created series with a 201 status code, handling any errors along the way.

When you think the route is ready, run the testing suite to check your work.

**50.**

Add a route param for :issueId. This param should check to see if an issue with the supplied issue ID exists. If so, it should continue to the next middleware function in the middleware stack. If not, it should return a 404 response.

**51.**

Add a PUT /:issueId route. This route should return a 400 response if any required fields are missing (name, issueNumber, publicationDate, or artistId). Additionally, the route should return a 400 response if an artist with the provided artist ID does not exist. Otherwise, it should update the issue in the database and return the newly-updated series with a 200 status code, handling any errors along the way.

When you think the route is ready, run the testing suite to check your work.

**52.**

Add a DELETE /:issueId route. This route should delete issue with the specified issue ID from the database. The route should handle any errors and send a 204 response upon successful deletion.

When you think the route is ready, run the testing suite to check your work.

**53.**

Finally, let’s write our DELETE /api/series/:seriesId route. In **series.js**, add a DELETE /:seriesId route. This route should check to ensure the specified series has no related issues in the database. If it does, the handler should return a 400 response. If not, the handler should delete the series from the database and send a 204 response.

When you think the route is ready, run the testing suite to check your work.

**Seed Database**

**54.**

At this point, your API should be complete and X-Press Publishing should work as expected! Run your testing suite (everything should pass) and try out all of the functionality on the site. If at any point you need to create more data, you can run our seed file to add some fake data to your database. To do this, run node seed.js from the root directory of the project.

Congratulations, you created a complete API for a fully-operational internal tool. This is an awesome achievement!