1. Make sure Node is installed
   1. You can do this in your cmd prompt window by typing “node -v”
   2. Our “npm” statements won’t work in our terminal if we don’t

Express App Setup: <This stuff has already been done>

1. Create a folder for our entire project, then create a folder within it called “backend” or something (“server” would also work). Inside that folder create a file called “server.js”
   1. The name doesn’t technically matter, it should be something we can immediately identify as the entry point to our backend server
2. Create a package.json file in our backend folder. Do this by opening a new terminal and cd’ing into our backend folder. Run “npm init -y”
   1. This will allow us to keep track of dependencies and register custom scripts
   2. If you don’t cd into the backend, the package.json file will be created in the root folder, which we don’t want
3. Install the express package with “npm install express”
   1. You’ll see a package-lock.json file created, as well as a node\_modules folder
   2. That folder is where all our dependencies are stored; we don’t need to worry about that at all
4. Require the express package and register our app
   1. const express = require(‘express’);
   2. const app = express();
5. Listen for requests on a certain port number
   1. We use port 4000 in our tutorial
   2. You could also set up a .env file to house your port number of choice and (later in the tutorial), your mongoDB URI (environment variables) so that you can keep that information private if you intended to post an application to a github repository
      1. This file would then need to be added to your .gitignore file, and probably add instructions to your README file saying to put in a port number manually.
      2. I forgot to create a .gitignore file so I will not be doing this. If you were to, you’d also need a line saying “require(‘dotenv’).config(); at the top of your server.js file
   3. Run the backend server with “node server.js”
6. Alternatively we can use nodemon when running our server. npm install -g nodemon
   1. This makes it so that when we save changes, it automatically restarts the server for us, rather than having to restart it manually
7. Register an additional script in your package.json file called “dev”, with its corresponding command being “nodemon server.js”
   1. This isn’t technically necessary, but at least you know how to register scripts now if you want to for some reason
8. Have your app respond to requests by establishing routes
9. You can check if your app is working so far by running “npm dev” and typing “localhost:4000” in your browser
10. Set up some middleware to log our requests and their paths

-------------------------------------------------------------------------------------------------------------------------------

Set up routes

We could test our routes and database functionality with Postman, however I personally never bothered so we’re simply going to hope and pray everything we do is correct

1. Set up our routes, or API endpoints
   1. This lets us interact with our database in different ways using the API
   2. Create a new folder called “routes”, and in it a file called songRoutes.js
2. We should have routes to:
   1. Get all song documents
   2. Create (post) a new song document
   3. Get a single song document
   4. Delete or update existing song documents (we may not have time for this)
      1. For each of this you’d have an app request handler like “.get”, or “app.post” to create a new document
3. Access the app from our new file using the following lines:
   1. const express = require(‘express’);
   2. const router = express.Router();
   3. router.get(‘/’, () => {}); //and similar lines to act as our endpoints
4. Access the routes from our entry point by adding “module.exports = router;” to the bottom of the file, and adding “const songRoutes = require(‘./routes/songRoutes’);”
   1. The .js handle is not needed
5. Use the routes with “app.use(‘/api/songs’, songRoutes);”, deleting the sample route we had before
   1. When we fire a request to that route (<http://localhost:4000/api/songs>) we use the songRoutes to see which endpoint gets used and therefore which function we fire (ie. a request to <http://localhost:4000/api/songs/> would go to our first “get” route)
   2. For now, we’ll just be responding with a json string at each endpoint, but later these will match up with functions in our controller file
6. Add “app.use(express.json());” to your middleware
   1. This ensures there’s some data in the request

—----------------------------------------------------------------------------------------------------------------------------

Set up your mongoDB connection

1. Create a mongoDB account
   1. Just go with the free option for now
   2. Once you make one, it \*should\* either bring you to a screen with a “build a database” button or just immediately try to get you to make one, I don’t remember which, either way, go ahead and create one (go with the free option), name your cluster
   3. It should then give you a popup with options on how to connect to the database. I \*believe\* the options you need to choose are “Connect your application”, then copy the connection string it provides.
2. If you’re using a .env file, in a new row type “MONGO\_URI=” and paste in the connection string without any spaces, otherwise just paste the connection string where I tell you to put process.env.MONGO\_URI
3. Install mongoose with npm install mongoose (cancel out of any other processes you have going on)
   1. This wraps mongoDB in another layer allowing us to use methods to read/write db documents, as well as declare models and schemas to enforce rules and structure on our documents. On a larger scale, you may not necessarily want this, it kinda depends on how you intend to process your data
4. Require mongoose on our server.js file with “const mongoose = require(‘mongoose’);”
5. Add a block to connect to your database:
   1. mongoose.connect(process.env.MONGO\_URI).then(() => {<listen for requests block goes here>}).catch((error) => { console.log(error) });
   2. Because we wouldn’t want to listen for requests until we’ve connected to the database, move the block for listening for requests into the {} in the “then” block

—----------------------------------------------------------------------------------------------------------------------------

Create a model and controller, and edit our routes

1. Create a schema for our data to follow. Create a models folder with a songModel.js file
   1. This ensures all of our data going into our collection has the same predictable structure
   2. Require mongoose
      1. MongoDB alone is schema-less
2. Declare a songSchema
   1. title: { type: String, required: true },
   2. artist: { type: String, required: true },
   3. album: { type: String, required: true }
3. Add a timeStamps property and set it to true
   1. This automatically creates this property on the documents for us, telling us when the doc was created
4. Export the model
   1. module.exports = mongoose.model(‘song’, songSchema);
      1. The collection name is automatically pluralized
   2. We are able to interact with our collection through this model; if we wanted to get all of the songs in our collection, we’d do something like songModel.find()
5. Create a controllers folder and songController.js
   1. Require songModel (‘../models/songModel’), and mongoose
   2. Add the functions getSongs, getSong, and createSong, all following the format below
      1. const getSongs = async (req, res) => {}
   3. For getSongs
      1. declare a const songs equal to songModel.find({}).sort({createdAt: -1});
      2. No parameters for find means we retrieve all
      3. Find is an asynchronous function, and we can’t do anything until it’s complete, so we need to wait for it to finish with “await”
      4. Sorting the way we are lists all the songs in descending order
      5. Set the response: res.status(200).json(songs);
         1. Status 200 is a good status, basically meaning nothing went wrong
         2. Turns songs into a json object
   4. For getSong
      1. get the id from the request parameters [req.params]: const {id} = req.params;
      2. We need to ensure the id would theoretically be correct first, with: if(!mongoose.Types.ObjectId.isValid(id)). If it’s not, then return a response status of 404 with the json object {error: ‘whatever error message you want’}
         1. 404 is a bad status, usually indicating something is not found
      3. Otherwise, set a const song to songModel.findById(id);
      4. If there is no song (!song), res.status(404) like before, otherwise repeat 27c.v., just with song instead of songs
   5. For createSong
      1. get the title, artist, and album from the request body the same way we did the id
      2. Do it all at once {title, artist, album}, don’t do it on separate lines
      3. Try to set a const song = songModel.create({title, artist, album}); //also an asynchronous function
      4. Set the response status accordingly like before, including in the catch block
6. Export these methods like so:
   1. module.exports = { createSong, getSongs, getSong }
7. Import these into your songRoutes similarly:
   1. const { createSong, getSongs, getSong } = require(‘../controllers/songController’);
8. Change all your routes
   1. The second parameter for each of these routes can simply be the name of the corresponding function from the controller

—----------------------------------------------------------------------------------------------------------------------------

Set up React app (Most of this is done for you)

1. Cancel out of any terminal processes and cd out with cd.. Then create a react app called frontend with: npx create-react-app frontend
   1. cd into it for later
   2. Delete everything in the src folder except for App.js, index.css, and index.js
2. Run npm install react-router-dom
   1. This allows us to navigate across our webpage, which would be useful if we had more than one page. This is outside the scope of our demo, but if you choose to go through the playlist I based this on, you’d actually need this
3. While we’re at it, run npm install react-bootstrap
   1. In hindsight, I don’t think this was actually necessary, but my team used it, and I had intended on basing my components off of it instead of whatever the videos made, so I installed it just in case
4. Edit the reportWebVitals import from index.js as well as everything below line 11, since we deleted the corresponding file
5. Delete all the content inside the div in App.js, as well as both imports
   1. Replace it with an import of { BrowserRouter, Routes, Route } from ‘react-router-dom’
   2. Inside the div, add an opening and closing <BrowserRouter> tag
      1. Between those, add a div with a className=”pages”
      2. Between those, add an opening and closing <Routes> tag
      3. Between those, add a self-closing <Route /> tag, with two different prompts
         1. path=”/”
         2. element={}
            1. This is where we’d add information as to what page this path goes to; we now need to make that page
6. Create a new folder pages, and inside create a file called “Home.js”

<This is the point where we start doing things ourselves>

1. Add this page’s content, we’re essentially creating a blank react component
   1. Const Home = () => {}
      1. Inside the curly brackets return some content: return(<content goes here>);
         1. For now, just add <div className=”home”><h2>Home</h2><h5>Submit a Song><hr /></div> but actually formatted properly
   2. export default Home;
      1. Now we can use this component and add it as the element for that route back on App.js It should look like: element={<Home />}

—----------------------------------------------------------------------------------------------------------------------------

Create Components to fetch and display the data

1. Create a components folder
   1. It’s important to note that all of your components need to start with a capital letter
2. Add the files: SongCard.js, SongContainer.js, and SongForm.js
3. We’ll start with SongContainer.
4. Import React, { useState, useEffect } from ‘react’;
5. Import SongCard from ‘./SongCard’;
6. Start building your SongContainer component = () => {}
   1. Set const[songs, setSongs] = useState([]);
      1. This means that the list of songs will be empty by default, we’ll be filling it soon
   2. Add a useEffect(() => {}, []); statement
      1. For the purposes of this application, the square brackets are not important, you could pass along some sort of parameter to render the component dynamically, but we have no such data to do this with
   3. We need to fetch the songs from the database
      1. Copy this into the curly brackets:

**const** fetchSongs = **async** () **=>** {

**const** response = await fetch('/api/songRoutes');

**const** json = await response.json();

if (response.ok) {

setSongs(json);

}

};

fetchSongs();

* + 1. We turn the response into a json object that can then be used to fill the songs list. This function is called as part of our useEffect statement.
    2. You’ll notice we don’t have the full backend address there, that is because we need to do some extra setup to allow cross-port communication, as our frontend is running on port 3000, and our backend on port 4000.
       1. In your frontend’s package.json file, not the backend’s, add this line to the top: “proxy”: “<http://localhost:4000>”,
       2. This tells our application to send any requests it doesn’t recognize to port 4000, where our backend is listening for requests
  1. Return the Container’s content: return (<Container content goes here>)
     1. Have all the content within a div
        1. Add an <h1> header, followed by another div with classname=”songs-container”
        2. Follow that with this block:

{songs && songs.map((song) **=>** (

<SongCard key={song.id} {...song} />

))}

Ignore the weird coloring, some shenanigans happened to me as I prepared this. For each song in the list, a card is rendered with the data from each song.

* + - 1. Below that add this block of css:

<style jsx>{`

.songs-container {

display: flex;

flex-wrap: wrap;

justify-content: space-around;

margin: 20px;

border: '1px solid #ccc';

padding: '15px';

margin: '10px';

borderRadius: '8px';

boxShadow: '0 4px 8px rgba(0, 0, 0, 0.1)';

backgroundColor: '#fff';

}

`}</style>

* + 1. Close both divs
  1. export default SongContainer;

1. Add the SongContainer to the home page: <SongContainer />
   1. Be sure to also import it

Next we’ll do the SongCard.

1. import React, { useState } from ‘react’;
2. Start building your SongCard component = ({ title, artist, album }) => {}
   1. Add this css block:

**const** cardStyle = {

border: '1px solid #ddd',

borderRadius: '8px',

padding: '16px',

margin: '8px',

boxShadow: '0 4px 8px rgba(0, 0, 0, 0.1)',

backgroundColor: '#fff',

};

* 1. Create a: function renderData() {}, and within it, return the card content: return(<card content goes here>);
     1. The reason we’re doing this one a little different is because this would make adding the capability of editing data much easier, however we lack the time to actually do so.
        1. Place the content within a div with style={cardStyle}
        2. Set an <h2> header with the {title}, followed by <p> paragraphs detailing who the song is by and what album the song is from
  2. Return the content but for realsies this time: return(<card content goes here>);
     1. This is where we return the content the way we do on the other two components
     2. Just place this block in: <div>

{renderData()}

</div>

* 1. export default SongCard;

—----------------------------------------------------------------------------------------------------------------------------

Create a form to submit data to the database

1. Create a SongForm.js file
2. import React, { useState } from ‘react’;
3. import { Alert } from ‘react-bootstrap’;
4. Start building your SongForm component = ({}) => {}
5. Within the component:
6. Declare a title, artist, album, and error constant with matching “set” methods, and set them equal to useState(‘’);
   1. They should match this format: const [title, setTitle] = useState(‘’);
7. Within the component, return the form content
   1. return ( <form content goes here> );
   2. All of your form content should go between <form> tags, on your opening tab, indicate an onSubmit method {handleSubmit}
   3. Have a div for each field
   4. Label each text field
      1. Between the opening and closing label tags, have the label text (duh), and a self-closing input tag declaring its type as text, the value as the corresponding constant, and the onChange method, it should look something like this:

<input type=”text” value={title} onChange={(e) => setTitle(e.target.value)} />

* 1. Add this line below the last div: {error && <Alert variant="danger">{error}</Alert>}
  2. Add a button of type=”submit” to the bottom of the form with text of your choice

1. Create a function for submission handling, it’ll be async with event as its parameter
   1. Prevent a default submission with event.preventDefault();
   2. Set a const song = {title, artist, album}
   3. Set a const response = await fetch(‘/api/songRoutes’, method: ‘POST’, body: JSON.stringify(song), headers: { ‘Content-Type’: ‘application/json’ } });

POST is the type of request, which helps our backend know which route to use

Our request body is a json string of the data being submitted

* 1. Set a const json = await response.json();
  2. If the response is not okay (!response.ok), set the error to the error from the json object. Otherwise, reset the title, artist, and album fields for a new submission, set the error to null (in case an error was already being rendered, and add a console log statement saying the song was created successfully, followed by the json string

1. export default SongForm;
2. Add the form to your home page

You can run your frontend with npm start

—----------------------------------------------------------------------------------------------------------------------------

Congratulations! You’ve completed this basic MERN stack application!