Exploration — Implementing REST APIs using Express

Introduction

We now describe how we can implement all the CRUD operations using HTTP methods in a RESTful Express app.

Example: REST API

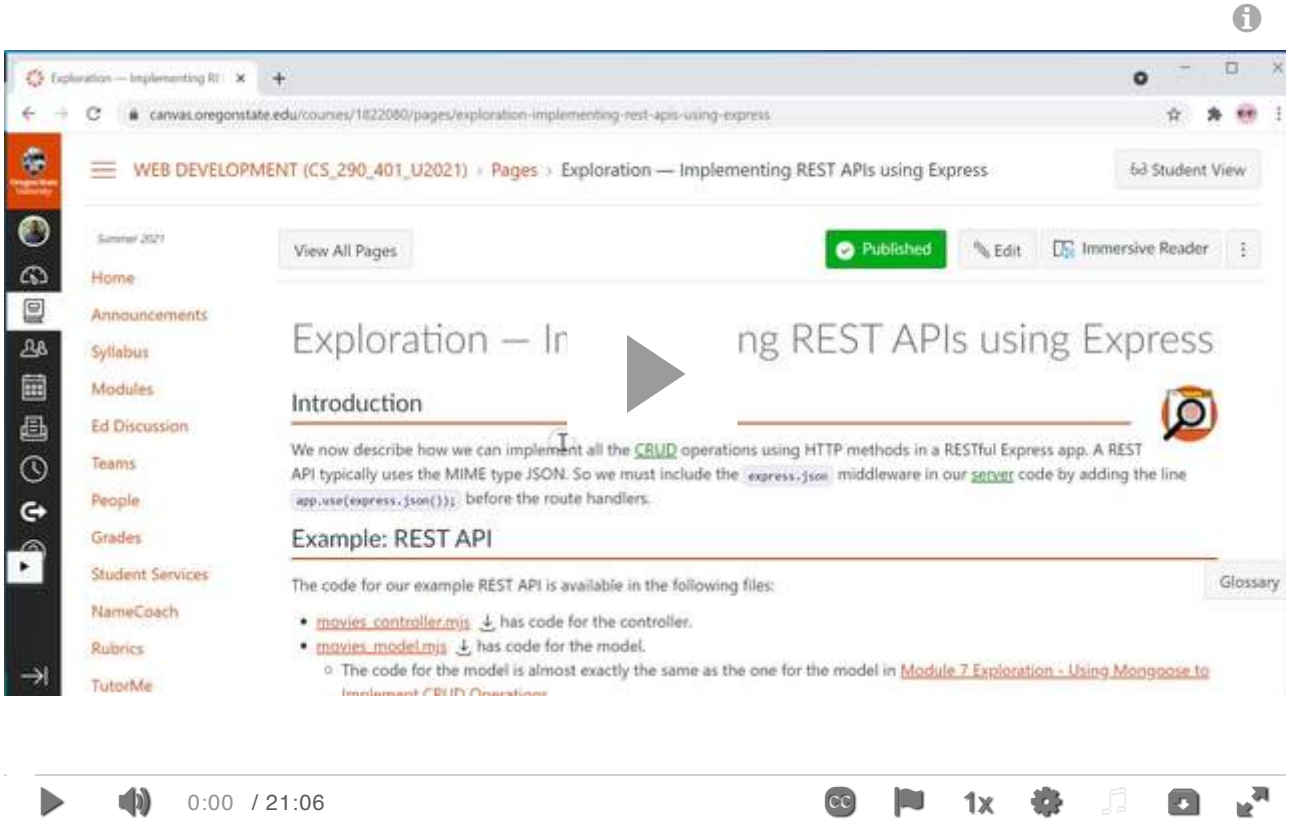
Download the **movies-api-backend-starter code (https://canvas.oregonstate.edu/courses/1879154/files/94538869?wrap=1)** (https://canvas.oregonstate.edu/courses/1879154/files/94538869/download?download_frd=1) and run npm install and npm start to explore and test REST API for use with the full stack app. This 'backend' api will be used in conjunction with the 'frontend' React app which provides a user-interface for the full-stack.

- movies-model.mjs **(https://canvas.oregonstate.edu/courses/1879154/files/93831985?wrap=1)** is nearly the same as for Module 7. Just a few updates are noted in the explanations, below.
- movies-controller.mjs has many changes from Module 7's code, which are explained below.
- package.json, which installs these dependencies: dotenv, Express, Mongoose, and Nodemon.
- .env
  - The file has 2 parameters,
    - PORT which you don't need to change (3000)
    - MONGODB_CONNECT_STRING which you used in the previous Module (either your localhost:27017 or Atlas Cluster URL.

Use one or more of these test files to see how the POST, GET, DELETE, and PUT code works with the MongoDB /movies collection:

- test-requests-video.http to send requests using the VS Code REST Client Extension (correlates with the video above).
- test-requests-more.http to send requests using the VS Code REST Client Extension (includes more CREATE requests and an additional filter request).
- test-requests.ps1 to send requests from Windows PowerShell using `Invoke-Request`.
- test-requests-curl-linux.sh to send requests from the MacOS or Linux shell using `curl`.
- test-requests-curl-windows.bat to send requests from WSL (Windows Subsystem for Linux) using `curl`.

The following video walks through how these files work together. Timestamps for the beginning of each of the CRUD topics are noted in the headings.



> **Note:** When the video was recorded, MongoDB had a command line interface called `mongo`. `mongo` has now has been superseded by `mongosh`, which we discussed in Module 7.

Middleware

A REST API typically uses the MIME type JSON. So we must include the `express.json` middleware in our server code by adding the line `app.use(express.json());` before the route handlers.

Create Using POST (2:11)

1. Request:
   - The client requests creation of a resource by sending a request with the HTTP method `POST`.
   - The body of the request contains a JSON object with the properties of the new resource.
2. Defining the route:
   - The route is defined using the Express method `app.post`.
   - The route handler gets the properties of the resource to be created from `req.body`.

- The route handler calls the relevant method in the model layer to create the resource.
- The model layer returns a promise that resolves to the newly created resource, including the unique ID of this resource.

3. Response:
   - The response includes the newly created resource with all its properties including the unique ID.
   - The status code is 201.

Example: Create Using POST (6:30)

Here is the route handler for the movie-controller.mjs file's `POST` requests to the URL `/movies`. The /movies path is the collection name.

- This route handler gets the values of the properties `title`, `year` and `language` in the JSON object sent in the body of the request.
- It calls the method `movies.createMovie()` in the model layer to create a new movie.
- In the response, it sends the JSON object returned by `movies.createMovie()` and sets the status code to 201.

```
app.post('/movies', (req, res) => {
    movies.createMovie(req.body.title, req.body.year, req.body.language)
        .then(movie => {
           // Confirm the creation
           res.status(201).json(movie);
        })
        .catch(error => {
            console.error(error);
            // Send a message if the request had an error.
            res.status(400).json({ Error: 'Request failed' });
        });
});
```

Testing for create is noted in the move at timestamp 3:24.

Read/Find Using GET (6:42)

There are few cases to consider, including:

1. Reading a single resource by its unique ID.
2. Reading the whole collection.
3. Reading a subset of the collection based on some criteria.

All these cases use the `GET` HTTP method.

Case 1: Reading a Single Resource by Its Unique ID (6:42)

Let us first consider the first case:

1. Request:
   - To read a specific resource, the client sends a request with the HTTP method `GET`.
   - The URL includes the ID of the resource, e.g., `http://localhost:3000/movies/60cfa8df411c4d66e8abcdbd`.
2. Defining the route:
   - The route is defined using the Express method `app.get`.
   - The route handler gets the ID of the resource from `req.params`.
   - The route handler calls the relevant method in the model layer to find the resource by ID.
   - The model layer returns a promise that resolves to the resource if the resource is found. Otherwise, the promise resolves to the null value.
3. Response:
   - If the resources is found, it is sent in the body as JSON with the status code 200.
   - If the resource is not found, the status code is 404.

Example: Read Using GET

Here is the route handler for the movie-controller.mjs file's `GET` requests from the URL `/movies`.

- This route handler gets the values of the path parameter `_id` from the request URL.
- It calls the method `movies.findMovieById()` in the model layer to get the movie with this ID value.
  - In the movies-model.mjs file, update the `findById` function so it references `findMoviebyID`, and, at the bottom of the file, update the export name to match.
- In the response, it sends the JSON object returned by `movies.findMovieById()`.
- However, if the model returns null, then the response is sent with status code 404.

```
app.get('/movies/:_id', (req, res) => {
    const movieId = req.params._id;
    movies.findMovieById(movieId)
        .then(movie => {
           if (movie !== null) {
           res.json(movie);
        } else {
           res.status(404).json({ Error: 'Resource not found' });
})
        .catch(error => {
            res.status(400).json({ Error: 'Request failed' });
        });

});
```

Case 2 and 3: Reading the Collection or Reading a Filtered Subset of the Collection (12:42)

- To read the whole collection, the URL will have just the name of the collection without an ID.

- To read a subset of the collection based on some criteria, these criteria can be specified as query parameters to the collection URL.

Example: Reading the Collection or Reading a Subset of the Collection

Here is the route handler for the movie-controller.mjs file's `GET` requests to the URL `/movies`.

- This route handler can return either the whole collection or return just the movies for the year provided in the request URL as a query parameter.
- The route handler looks to see if the query parameters include a parameter with the name `year`.
  - If this parameter is found, the route handler creates a filter on the property `year` and the value from the query parameter.
  - If this parameter is not found, the route handler creates an empty filter which is equivalent to not specifying any filter.
- It calls the method `movies.findMovies()` in the model layer to query the collection.
- In the response, it sends the JSON object returned by `movies.findMovies()`.

```
app.get('/movies', (req, res) => {
    let filter = {};
    // Is there a query parameter named year? If so add a filter based on its value.
    if(req.query.year !== undefined){
        filter = { year: req.query.year };
    }
    movies.findMovies(filter, '', 0)
        .then(movies => {
            res.send(movies);
        })
        .catch(error => {
            console.error(error);
            res.send({ Error: 'Request failed' });
        });

});
```

Update Using PUT (14:33)

1. Request:
   - To update a resource, the client sends a request with the HTTP method `PUT`.
   - The URL includes the ID of the resource, e.g., `http://localhost:3000/movies/60cfa8df411c4d66e8abcdbd`.
   - The body of the request contains a JSON object with all the non-ID properties to set on the resource.
2. Defining the route:
   - The route is defined using the Express method `app.put`.
   - The route handler gets the ID of the resource from `req.params`.
   - The route handler gets the properties of the resource to be updated from `req.body`.
   - The route handler calls the relevant method in the model layer to update the resource.

- In the movies-model.mjs file, update the `updateMovie` function so it references `replaceMovie`, and, at the bottom of the file, update the export name to match. (timestamp 16:46)
- The model layer returns a promise that resolves to the number of documents that were updated.
- If a resource with the specified ID exists, then the this value is 1. Otherwise, the value is 0.

3. Response:
   - If the resources is found and updated, it is sent in the body as JSON with the status code 200.
   - If the resource is not found, the status code is 404.

Example: Update Using PUT

Here is the route handler for the movie-controller.mjs file's `PUT` requests to the URL `/movies`.

- This route handler gets the values of the path parameter `_id` from the request URL.
- The route handler gets the values of the properties `title`, `year` and `language` from the JSON object sent in the body of the request.
- It calls the method `movies.replaceMovie()` in the model layer to update this movie.
- In the response, it sends the JSON object returned by `movies.replaceMovie()`.

```
app.put('/movies/:_id', (req, res) => {
    movies.replaceMovie(req.params._id, req.body.title, req.body.year, req.body.language)
        .then(numUpdated => {
            if (numUpdated === 1) {
                res.json({ _id: req.params._id, title: req.body.title, year: req.body.year, la
nguage: req.body.language })
            } else {
                res.status(404).json({ Error: 'Resource not found' });
            }
        })
        .catch(error => {
            console.error(error);
            res.status(400).json({ Error: 'Request failed' });
        });
});
```

Delete Using DELETE (17:29)

1. Request:
   - To delete a resource, the client sends a request with the HTTP method `DELETE`.
   - The URL includes the ID of the resource to delete, e.g., `http://localhost:3000/movies/60cfa8df411c4d66e8abcdbd`.
2. Defining the route:
   - The route is defined using the Express method `app.delete`.
   - The route handler gets the ID of the resource from `req.params`.
   - The route handler calls the relevant method in the model layer to delete the resource.

        - Only `deleteById` will be needed for this app; `deleteByPropery` isn't needed.
    - The method in the model layer returns a promise that resolves to the count of deleted resources.
        - This value is 1 if a resource with the given ID existed.
        - This value is 0 if no resource had the given ID.
3. Response:
    - If the resource was found and deleted, the response status code is 204 indicating that the resource was successfully deleted.
    - If the resource was not found, the response status code is 404.

Example: Delete Using DELETE

Here is the route handler for the movie-controller.mjs file's `DELETE` requests to the URL `/movies/:_id`.

- This route handler gets the values of the path parameter `_id` from the request URL.
- It calls the method `movies.deleteById()` in the model layer to update this movie.
- In the response, it sends either status code 204 if a resource was deleted by the call to `movies.deleteById()` or status code 404 if the resource was not found.

```
app.delete('/movies/:_id', (req, res) => {
    movies.deleteById(req.params._id)
        .then(deletedCount => {
            if (deletedCount === 1) {
                res.status(204).send();
            } else {
                res.status(404).json({ Error: 'Resource not found' });
            }
        })
        .catch(error => {
            console.error(error);
            res.send({ error: 'Request failed' });
        });
});
```

Summary

In this exploration, we went over a detailed example of implementing a REST API in Express. Before we move on, we want to note that there are some details about REST APIs that we did not get into. For example, many REST APIs also support the HTTP method `PATCH` for partial update of a resource. There are also variations in what is returned by `POST`, `PUT` and `DELETE` methods. However, the material covered in this exploration should give you a fairly good idea of what REST APIs do and how you can go about implementing them.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Apigee, a company that Google bought a few years back, has a good **free book on REST API design (https://pages.apigee.com/web-api-design-register.html)** .
- We have not discussed how to secure REST APIs. Auth0, a company which provides products for secure access, has a **tutorial (https://auth0.com/blog/node-js-and-express-tutorial-building-and-securing-restful-apis/)** that describes how to develop REST APIs using Node and Express, and secure them using Auth0.