

## Overview

This coursework tasked us with creating a RESTful (Representational State Transfer) Web API using Express.js that can manipulate media store entities using a promise-based CRUD interface. We were to implement various functions of input validation in addition to several endpoints, and provide appropriate testing using supertest and jest.

## Design

### **ValidationHandler**

I began by working on my input validation, as per the first requirement. I have a method to validate the argument length, printing an error message and exiting the program if it is not equal to 3. The path is then taken from the third argument and resolved to an absolute file path to avoid the ambiguity of using relative file paths. If the file does not exist, an error message is printed, and the program is terminated.

Then, I created a method `parseFile` to read the file and return the parsed JSON data, printing an error message and exiting if there was an error in parsing (such as if the data is not in the JSON format).

The specification states that the data set must be valid JSON and satisfy data set field constraints. Therefore, I made several methods to achieve this functionality. My `validateASCII` method takes a string as a parameter and uses regular expressions to return a boolean representing whether the string only contains ASCII characters or not, and my `hasValidFields` method verifies whether the data satisfies the field constraints. These methods are called within the `isValidObject` method, which also checks whether the data contains all the query fields. This method takes the object as a parameter, but also takes a boolean `isInitialLoad`, which denotes whether an error message is to be printed to the standard output or not. When reading from the file, this is set to true, but when validating objects for 'POST' or 'PUT' requests, it is set to false.

Finally, my `populateStore` method takes a JSON dataset and validates the data. It uses a for loop to iterate through each object and uses a destructuring assignment to define variables, which are then passed to the `create` function of the `MediaStore` object. This returns a promise which is awaited and resolved with an ID of the newly created media object, which is added to a set of IDs.

I made the decision to have the method load objects in order, as I found this to be most logical as it allows for consistent results for testing. This means that the program awaits a promise to be resolved for each of the objects and comes at the cost of the program taking  $(20 \times 31 = 620)$  milliseconds, worst case, to run as the timeout is between 4 and 31ms and there are 20 objects being loaded.

These methods are then called in a run method, which returns the set of created IDs. I have chosen to store the generated IDs in a set to make use of idempotency, allowing my program to check whether a media object exists in the store or not and return a not-found error faster than if it were using promises. My implementation does not cache the data itself, such as in a key-value pair, but rather tracks the existence of the object in the store, to conserve space.

Initially, I had written all of my code in one file, but as I worked on more requirements, I realised that separating my code into different files would be beneficial for testing and readability, and so I divided them into a class for handling validation and store population, a class for handling the server, and a main index file that imports these classes.

## **ServerHandler**

In my ServerHandler class, I import ValidationHandler and instantiate it in the constructor. I created a constant formattedMediaObject which uses an arrow function to format objects to a specific layout. My constructor takes a set of IDs and a store object and instantiates defaultLimit, defaultOffset, and creates an instance of the ValidationHandler class.

Then, I made a getMedia method to improve code readability and reduce code repetition by retrieving and formatting a media object, taking an ID as a parameter and returning the formatted object.

I used dependency injection for more flexibility while testing, and this is done by having the ServerHandler take a MediaStore object as input and having the createServer method return a server, which is then used to listen on a port.

I have a createServer method which creates an Express web application to handle HTTP requests and defines routes. I added the express.json middleware to parse incoming JSON payloads and set the request.body attribute to the parsed data. Then, I included error-handling middleware for SyntaxError errors, which returns a status code of 400 if the object sent in a 'PUT' or 'POST' request is not in the JSON format.

My 'GET' endpoint at the '/media' route allows the user to retrieve all media objects. This is done by extracting the limit and query parameters, and filtering the retrieved objects based on whether they match the name, type, and description or not. If the URL contains a name, type, or description query, a variable is assigned the boolean of whether the query matches any objects or not. If there is no query parameter, this value defaults to true, meaning that it matches all objects. This implementation allows for URL encoding characters as I utilise the decodeURIComponent method, and it is not case sensitive as I have used the toLowerCase method.

The length of the filteredObjects array is stored in a totalCount variable. If the value is 0, a status code of 204 is sent, indicating that the list in the responses is empty. Further checks are conducted to ensure that the limit and offset provided are greater than 0, and that the offset is less than the total count. If any of these returns "false", the program returns a status code of 500.

Then, the `filteredObjects` array is segmented using the `slice` method, based on the offset and limit values, and mapped to the `formattedMediaObject`. The URLs for “next” and “previous” are constructed depending on the values of the limit and offset provided. If the sum of the limit and offset is greater than the total number of objects, the value of “next” is null. If not, the new offset is the sum of the current limit and offset.

If the offset is 0, “previous” is set to null. If the offset is greater than 0, the URL for “previous” is set to have the same limit and an offset of  $(\text{offset} - \text{limit})$ . If the current offset is less than the query limit, however (meaning that the new offset would be negative), the new limit becomes the value of the current offset, and the new offset is 0. While I could have used separate if statements to segment these comparisons into different parts, I chose to nest the ternary operators within the URL as I found it less cluttered, more concise, and easier to work with. If there is no limit or offset provided in the query, the values default to `defaultLimit` and `defaultOffset`.

The response is then formatted and sent with the status code of 200. This entire process is encapsulated within a try-catch block, which returns a status code of 500 if there are any unexpected errors.

My ‘GET’ endpoint at the ‘`/media/:id`’ route extracts the ID from the route handler parameters object and converts it to a Number. It checks if the ID set contains the requested ID, returning a status code of 404 if it does not. If it does, it retrieves the media object with the requested ID and formats it, returning a status code of 200 alongside a JSON response containing the formatted object.

The ‘PUT’ endpoint at the ‘`/media/:id`’ route updates an existing media object. It extracts the ID and converts it to a Number and uses the `req` object to access the request body. It verifies whether the data being sent is valid or not by calling the `isValidObject` method on the instance of the `ValidationHandler` class. If it is not valid, it returns a status code of 400. Then, it verifies whether the ID of the object to be updated exists in the set, returning a status code of 404 if it is not found. If it exists, it uses the `update` method on the store object and retrieves the updated object, formatting it and sending it alongside a status code of 200. If there is an error, the server sends a status code of 500, as everything is enclosed within a try-catch block.

Next, I implemented a ‘DELETE’ endpoint on the ‘`/media/:id`’ route which allows the user to remove a specific media object from the store. Once again, the ID is extracted and converted to a Number, and the server checks whether the ID exists in the set of IDs. If it does not, it returns a status code of 404. If it does, the `delete` function is called on the store object, using the requested ID. The ID is removed from the set, and the server returns a status code of 204. This functionality is surrounded by a try-catch block, which sends a status code of 500 if there are any errors.

I have a 'POST' endpoint at the '/media' route, which allows for media object creation. It validates whether the provided data is formatted correctly, returning a status code of 400 if it is not. If it is valid, it creates a new media object using the MediaStore's create function and adds the returned ID to the set of IDs. Then, it retrieves the newly created object, formats it, and sends it with a status code of 201. This is done within a try-catch block, which sends a status code of 500 if there is an error.

Finally, I have another 'POST' endpoint at the '/transfer' route, which allows users to transfer media from one server to another. I used regular expressions to ensure that the source is formatted correctly, returning a status code of 421 if it is not. The ID is extracted from the relative URL and compared to those in the ID set. If it does not exist, a status code of 404 is returned. If it does exist, the media object is retrieved, formatted, and sent to the target URL.

This is achieved using the node-fetch module to make HTTP requests to the target server. The request being made in this case is a 'POST' request, and the body of the fetch request is the object to be transferred. The response from the target server is then awaited, stored in a variable, and formatted by adjusting the ID to be the fully qualified URL for the ID. The object's ID for the source is then deleted from the ID set and the store, and the target's response is sent alongside a status code of 200. This code is wrapped in a try-catch block which sends a status code of 421 if the error is a FetchError, meaning that the target server does not exist, and a status code of 500 if not.

I chose to use async/await to make async requests rather than the "then" callback function, as I found it significantly easier to work with due to my familiarity with the code structure, and I wanted to minimise "promise chaining".

The index.js file imports the ValidationHandler and ServerHandler classes, which were exported in their respective files. It instantiates various objects of these classes and calls the run function of the ValidationHandler class to process the user's arguments. After processing the arguments and populating the store, it uses the returned ID set to create a server and bind it to a port using the listen function.

## Testing

To test my program, I accounted for various normal, edge, and exceptional test cases to ensure my program remained robust in different scenarios. I used the Supertest testing library of the Jest framework to test my program through unit tests. These tests attempt to cover, to an extent, integrated testing as well. This is demonstrated through the employment of different route handlers within one test to ensure they work as intended (such as getting a media object after deleting or transferring it).

Examples of edge cases include retrieving from an empty list or searching for a specific media object using search queries, and exceptional cases included invalid input and handling errors with a status code of 500.

I used a `beforeEach` and `afterEach` hook to reduce repetition in my testing, and to ensure proper setup and teardown of the servers. In my `beforeEach`, I created the server and made it listen on a specified port number, and in my `afterEach` I closed the server connection to the port. This likely contributes to the long runtime of the tests, as the `populateStore` method is being called before each test. To improve this, I could have had a `beforeAll` and `afterAll` hook, but I believe that it would have given me less flexibility in testing and thus chose to use `beforeEach` and `afterEach`.

My tests pass without any issues, as evidenced by the output:

Test Suites: 1 passed, 1 total

Tests: 48 passed, 48 total

Snapshots: 0 total

Time: 20.518 s

Ran all test suites.

I have provided tabulated descriptions of my tests below.

## GET

What is being tested	Purpose	Outcome
Pagination, status code 200.	Ensure that the endpoint can handle large datasets by providing paginated responses, allowing efficient retrieval of media objects.	(As expected) Status code 200.
Successful retrieval, status code 204.	Validate that the endpoint gracefully handles cases where no media objects match the query, returning a 204 status.	(As expected) Status code 204.
Paginated response returning first 2 media objects.	Confirm that the endpoint correctly paginates and responds with the requested number of media objects, supporting client-side pagination.	(As expected) Paginated response with the first 2 media objects.
(Offset + Limit) out of bounds.	Ensure that the endpoint responds appropriately when the offset and limit parameters exceed the total number of media objects.	(As expected) Paginated response with the remaining media objects.

"Previous" response with limit.	Verify that the endpoint correctly provides a "previous" link when the limit would cause the result set to go out of bounds.	(As expected) Previous link is provided in the response.
Error response.	Ensure that the endpoint responds with a 500 status when an error occurs during processing, indicating potential server issues.	(As expected) Status code 500.
Name query.	Confirm that the endpoint correctly filters media objects based on the provided name query parameter.	(As expected) Response with the media object matching the name query.
Type query.	Validate that the endpoint allows users to narrow down results by type.	(As expected) Response with the count of media objects matching the type query.
Desc query.	Ensure that the endpoint allows users to search for media objects based on their descriptions.	(As expected) Response with the media object matching the desc query.
All queries together.	Validate that the endpoint combines multiple query parameters, enabling users to perform complex searches.	(As expected) Response with the media object matching all queries.
Search queries with limit and offset.	Confirm that the endpoint handles search queries alongside pagination, supporting scenarios where users want to retrieve specific results in chunks.	(As expected) Paginated response with media objects matching the search query.
Queries with only the limit specified.	Verify that the endpoint correctly paginates when only the limit parameter is specified, providing flexibility for users who want to retrieve a specific number of results.	(As expected) Paginated response with the specified limit.
Queries with only the offset specified.	Confirm that the endpoint supports pagination when only the offset parameter is specified, accommodating scenarios where users want to start from a specific position.	(As expected) Paginated response with the specified offset.
Case-insensitive searches.	Ensure that the endpoint performs case-insensitive searches, enhancing the user experience by allowing flexible input.	(As expected) Response with the media object matching the case-insensitive search query.

URL-encoded query searches.	Validate that the endpoint correctly handles URL-encoded query parameters, supporting scenarios where users provide encoded input.	(As expected) Response with the media object matching the URL-encoded query parameters.
Successful media object retrieval with id.	Confirm that the endpoint returns a 200 status when successfully retrieving a specific media object.	(As expected) Status code 200.
Media object response	Ensure that the endpoint responds with the requested media object, supporting scenarios where users need detailed information about a specific media item.	(As expected) Response with the requested media object.
Non-existent media object.	Validate that the endpoint returns a 404 status when attempting to retrieve a non-existent media object, allowing clients to handle missing resources appropriately.	(As expected) Status code 404.
Error mode turned on.	Ensure that the endpoint returns a 500 status when an error occurs during processing with error mode enabled, signaling potential server issues.	(As expected) Status code 500.

## POST

What is being tested	Purpose	Outcome
Successful media addition.	Verify that the endpoint returns a 201 status when successfully adding new media, enabling clients to confirm the success of media creation.	(As expected) Status code 201.
New media object response.	Confirm that the endpoint responds with the newly added media object, facilitating scenarios where clients need information about the created media.	(As expected) Response with the transferred media object.
Field validation.	Validate that the endpoint returns a 400 status when receiving a request with	(As expected) Status code 400.

	incorrect or missing fields, guiding clients on proper payload structure.	
ASCII validation.	Ensure that the endpoint rejects requests with non-ASCII characters in fields, promoting data integrity and preventing potential encoding issues.	(As expected) Status code 400.
JSON validation.	Verify that the media object being sent is in the JSON format, returning a 400 status if it is not.	(As expected) Status code 400.
Error mode on.	Confirm that the endpoint returns a 500 status when an error occurs during processing with error mode enabled, signaling potential server issues.	(As expected) Status code 500.
Adding a media object with the TAPE type.	Verify that the endpoint allows for a TAPE type to be added.	(As expected) Status code 201.
Non-existent media object for transfer.	Confirm that the endpoint returns a 404 status when attempting to transfer a non-existent media object, allowing clients to handle missing resources appropriately.	(As expected) Status code 404.
Successful transfer.	Validate that the endpoint returns a 200 status when successfully transferring a media object, providing feedback on the success of the operation.	(As expected) Status code 200.
Transferred media object response.	Ensure that the endpoint responds with the transferred media object, allowing clients to obtain information about the transferred resource.	(As expected) Response with the transferred media object.
Incorrect target URL.	Confirm that the endpoint returns a 421 status when the target URL provided in the transfer request is incorrect or unreachable.	(As expected) Status code 421.
Incorrect source URL.	Validate that the endpoint returns a 421 status when the source URL provided in the transfer request is incorrect or invalid.	(As expected) Status code 421.
Transferring a media object that has already been transferred.	Ensure that the endpoint returns a 404 status when attempting to transfer a media object that has already been	(As expected) Status code 404.



	transferred, guiding clients to handle scenarios where a resource is no longer available.	
Error mode on.	Confirm that the endpoint returns a 500 status when an error occurs during processing with error mode enabled, signaling potential server issues.	(As expected) Status code 500.

**PUT**

<b>What is being tested</b>	<b>Purpose</b>	<b>Outcome</b>
Successful media update.	Verify that the endpoint returns a 200 status when successfully updating an existing media object, allowing clients to confirm the success of the update operation.	(As expected) Status code 200.
Updated media object response	Confirm that the endpoint responds with the updated media object, providing clients with the most recent information about the modified resource.	(As expected) Response with the updated media object.
Field validation.	Validate that the endpoint returns a 400 status when receiving a request with incorrect or missing fields during a media update, guiding clients on proper payload structure.	(As expected) Status code 400.
ASCII validation.	Ensure that the endpoint rejects requests with non-ASCII characters in fields during a media update, promoting data integrity and preventing potential encoding issues.	(As expected) Status code 400.
JSON validation.	Verify that the media object being sent is in the JSON format, returning a 400 status if it is not.	(As expected) Status code 400.
Non-existent media object.	Confirm that the endpoint returns a 404 status when attempting to update a non-existent media object, guiding clients to handle scenarios where the resource is not found.	(As expected) Status code 404.

Error mode on.	Confirm that the endpoint returns a 500 status when an error occurs during processing with error mode enabled, signaling potential server issues.	(As expected) Status code 500.
----------------	---	--------------------------------

**DELETE**

What is being tested	Purpose	Outcome
Successful deletion.	Verify that the endpoint returns a 204 status when successfully deleting an existing media object, indicating the successful removal of the resource.	(As expected) Status code 204.
Accessing deleted resource ID.	Ensure that the endpoint removes the specified media object from the store after a successful deletion operation.	(As expected) Media object is no longer available in the store, status code 404.
Non-existent media object.	Confirm that the endpoint returns a 404 status when attempting to delete a non-existent media object, guiding clients to handle scenarios where the resource is not found.	(As expected) Status code 404.
Error mode on.	Confirm that the endpoint returns a 500 status when an error occurs during processing with error mode enabled, signaling potential server issues.	(As expected) Status code 500.

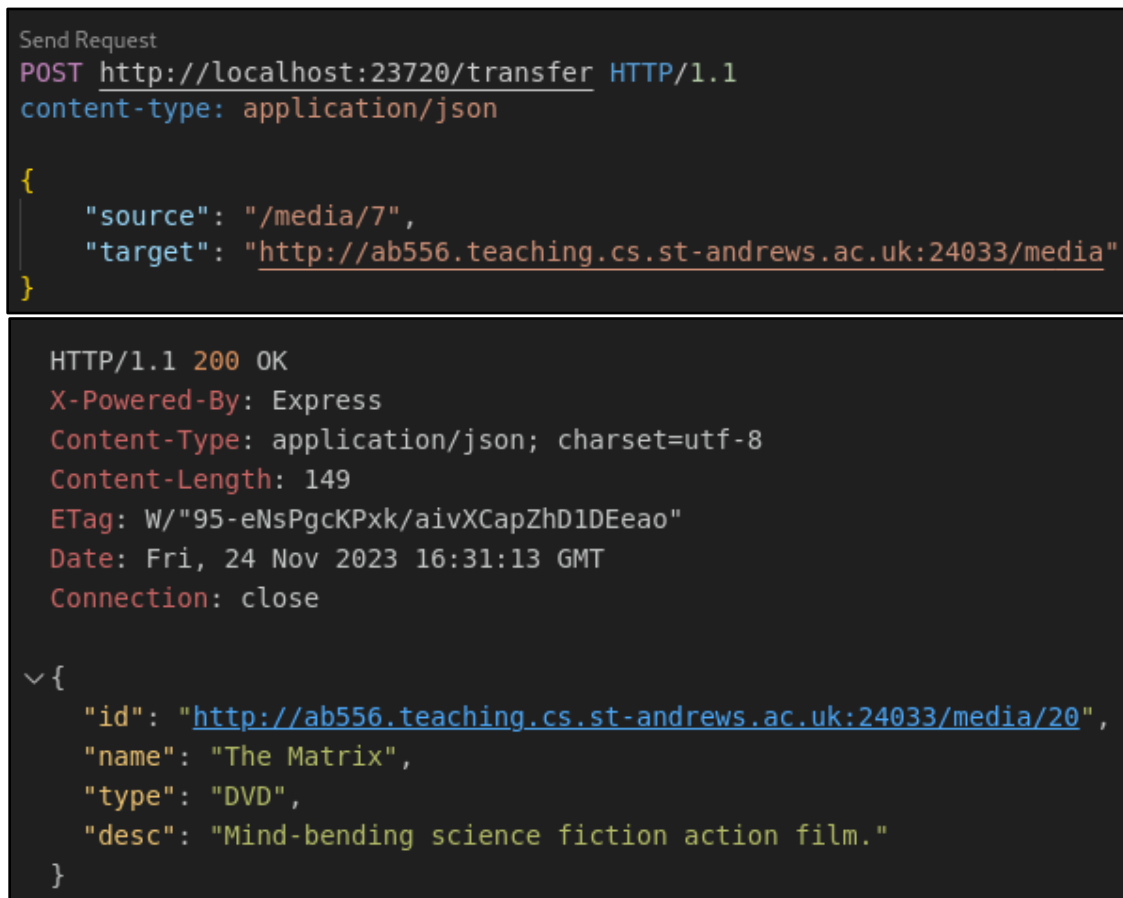
**Validation**

What is being tested	Purpose	Outcome
Incorrect argument length.	Ensure that the validation function correctly exits the process with status code 1 when provided with an incorrect number of command line arguments.	(As expected) Process exits with status code 1.
Incorrect file path.	Validate that the validation function correctly exits the process with status code 1 when provided with an incorrect file path.	(As expected) Process exits with status code 1.

ASCII functionality.	Confirm that the ASCII validation function correctly identifies non-ASCII characters.	(As expected) Returns false for a string containing non-ASCII characters
Functionality of method to populate store.	Ensure that the store population function correctly adds new media objects to the store and returns a set of unique IDs for the added objects.	(As expected) Set contains the expected IDs of the newly added media objects.

I attempted to test the `isValidObject` method using the `spyOn` mock function for `console.log`, but faced some issues and thus did not include these tests. However, I have tested this manually and they work as intended, outputting: “Invalid data. Please ensure all object fields are present.” if an object field is omitted; “Invalid data. Please ensure objects are formatted correctly.” if the constraints are not met; “Invalid data. Please only include ASCII characters.” if there are non-ASCII characters entered.

Some additional tests that were not included in the test suite involve parsing files, in which a non-JSON data set results in the program outputting “Error parsing file” and terminating the program. I have also added additional manual tests for requirement 10 to ensure the functionality is correct, using a peer’s server to send media objects. I have provided screenshots detailing this below.



```
Send Request
POST http://localhost:23720/transfer HTTP/1.1
content-type: application/json

{
  "source": "/media/7",
  "target": "http://ab556.teaching.cs.st-andrews.ac.uk:24033/media"
}

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 149
ETag: W/"95-eNsPgckPpk/aivXCapZhD1DEeo"
Date: Fri, 24 Nov 2023 16:31:13 GMT
Connection: close

{
  "id": "http://ab556.teaching.cs.st-andrews.ac.uk:24033/media/20",
  "name": "The Matrix",
  "type": "DVD",
  "desc": "Mind-bending science fiction action film."
}
```

Figure 1: Using ‘POST’ to transfer media to a target server.

```
Send Request
GET http://ab556.teaching.cs.st-andrews.ac.uk:24033/media/20

HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 102
ETag: W/"66-kvabBs7gKuEDxZ5CcxNdb4AUPEc"
Date: Fri, 24 Nov 2023 16:33:41 GMT
Connection: close

{
  "id": "/media/20",
  "name": "The Matrix",
  "type": "DVD",
  "desc": "Mind-bending science fiction action film."
}
```

Figure 2: Using 'GET' to retrieve transferred media from target server.

As shown in Figures 1 and 2, the program successfully posts media from the source to a target, and this can be verified using the GET endpoint, with the retrieved media corresponding with that which was transferred.

## Evaluation

My program successfully achieves all the requirements outlined for the practical, allowing a user to load JSON objects into a media store, and then manipulate the media through API requests. I have also extensively tested the different functions to ensure they work as intended in various scenarios.

## Conclusion

I found this practical to be very fascinating and quite different from our previous coursework; I enjoyed testing and adjusting my code to make it easier to test. I had some challenges with debugging in JavaScript, but once I became familiar with the syntax for the RESTful API, I found it much easier. I also found Jest and Supertest to be quite useful, as they helped me identify specific errors with my code which I would not have otherwise found.

Given more time, I would consider integrating the principle of idempotency more into the code, for example, to make it easier when users use 'PUT' to "update" a media object with the exact same information where in effect there would be no change. I would also consider using a logging mechanism to track incoming API requests and errors.

## References

1. [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)
2. <https://www.npmjs.com/package/supertest>
3. <https://jestjs.io/docs/jest-object>
4. <https://studres.cs.st-andrews.ac.uk/CS2003/Lectures/cs2003-web-09-api-design-rest.pdf>
5. <https://studres.cs.st-andrews.ac.uk/CS2003/Lectures/cs2003-web-10-express.pdf>