EECS 4313 - Software Testing

Cinaecloud Testing Report
Team E

Team members/contributors:-
Daivakshi Vaidya (217761016)
Chun-Kit Chung (217125329)
Ratul Momen (216254724)

Instructions for for to run Tests:

1. Setup environment
2. Define dependencies in package json file (for mongoose, mongodb-memory-server, etc)
3. For each test case, change login details (email and password) to your respective emails used for testing
4. Swap IDs with the right mongo instances or created instances (project ID, epk ID, etc)

Changes made to additional files to help smooth the testing process:-

1. Service files (eg for user controller spec file, we changed the user service spec file)
2. Incorrect imports in approx 60 files corrected
3. Env file and package.json file
4. Additional simple test files fixed to make them pass if app is functioning correctly

We have diligently complied with all the testing requirements and suggestions as per the outline. However, due to an issue with setting up the environment required for testing file-related functionality (e.g., file upload, delete, etc.), we were unable to carry out tests for the same in the file.controller.spec.ts file.

For each of the controller units that we have tested, we have attached a file. Please note, we have made several changes to the project files, please refer to the project file on github for more details. Please reach out to daivakshi@gmail.com incase of any confusion.

Link to project file on github: https://github.com/Daivakshi/cineacloud_backend_service_Testing

All the units tested are listed below:

1. User controller (Users)

File: users.controller.spec.ts

Test Case	**POST /login**
Description	This test case checks whether a user can log in with a valid email and password.
How it works	It sends a `POST` request to the `/users/login` endpoint with a valid email and password, then checks that the response status code is `200` and the response body contains a message saying "Logged in successfully" and a token.
Pass/Fail	Pass
Bugs	None

Test Case	**POST /login**
Description	This test case checks whether a user can log in with a valid email and password.
Expected	- Status code 200 - Message: “Logged in Successfully”
Actual	- Status code 200 - Message: “Logged in Successfully”

Test Case	**POST /login with wrong password**
Description	This test case checks whether a user can log in with an invalid password.
How it works	It sends a `POST` request to the `/users/login` endpoint with a valid email and password, then checks that the response status code is `400` and the response body contains a message saying "Logged in successfully" and a token.
Pass/Fail	Pass
Bugs	None
Expected	- Status code 400 - Message: “The username or password entered is invalid”
Actual	- Status code 400 - Message: “The username or password entered is invalid”

Test Case	**POST /login with wrong email**
Description	This test case checks whether a user can log in with an invalid email.
How it works	It sends a `POST` request to the `/users/login` endpoint with an invalid email and a valid password, then checks that the response status code is `400`.
Pass/Fail	Pass
Bugs	None
Expected	Status code 400
Actual	Status code 400

Test Case	GET /profile
Description	This test case checks whether a user can access their profile with a valid token.
How it works	It sends a `GET` request to the `/users/profile` endpoint with a valid token in the `Authorization` header, then checks that the response status code is `200`.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200
Actual	Status code 200

Test Case	GET /profile without token
Description	This test case checks whether a user can access their profile without a token.
How it works	It sends a `GET` request to the `/users/profile` endpoint without a token, then checks that the response status code is `401`.
Pass/Fail	Pass
Bugs	None
Expected	Status code 401 - unauthorised
Actual	Status code 401

Test Case	POST /createProject
Description	This test case checks whether a user can create a project with a valid token.
How it works	It sends a `POST` request to the `/users/projects` endpoint with a valid token in the `Authorization` header and a project name, then checks that the response status code is `200` and the response body contains a message saying "Project created successfully".
Pass/Fail	Pass
Bugs	None

Test Case	POST /createProject
Description	This test case checks whether a user can create a project with a valid token.
How it works	It sends a `POST` request to the `/users/projects` endpoint with a valid token in the `Authorization` header and a project name, then checks that the response status code is `200` and the response body contains a message saying "Project created successfully".
Expected	Status code 200
Actual	Status code 200

Test Case	POST /login
Description	This test case checks whether a user can create a project with a valid token.
How it works	It sends a `POST` request to the `/users/projects` endpoint with a valid token in the `Authorization` header and a project name, then checks that the response status code is `201`.
Pass/Fail	Pass
Bugs	None
Expected	Status code 201
Actual	Status code 201

Test Case	POST /login
Description	This test case checks whether a user can get their projects with a valid token.
How it works	It sends a `GET` request to the `/users/projects` endpoint with a valid token in the `Authorization` header, then checks that the response status code is `200` and the response body contains a message saying "User Projects".
Pass/Fail	Pass
Bugs	None

Test Case	POST /login
Description	This test case checks whether a user can get their projects with a valid token.
How it works	It sends a `GET` request to the `/users/projects` endpoint with a valid token in the `Authorization` header, then checks that the response status code is `200` and the response body contains a message saying "User Projects".
Expected	Status code 200
Actual	Status code 200

Test Case	POST /login
Description	This test case checks whether a user can get a project with a valid token and a valid project id.
How it works	It sends a `GET` request to the `/users/projects/:id` endpoint with a valid token in the `Authorization` header and a valid project id, then checks that the response status code is `200` and the response body contains the project information.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200 “OK”
Actual	Status code 200

What techniques you have used for testing?

Some of the techniques we used to test this unit were Boundary Value Testing and Unit Testing. We check the validity of the email and password inputted from the user and return the expected output for each test case. Integration testing is used to test the interaction between different components of the system, such as testing the interaction between the UserController and the NestJS application. On the other hand, unit testing is used to test individual units of the system, such as testing the UserController functions.

2. Props

File: props.controller.spec.ts

1. POST /props/create:

Test Case	**POST /props/create:**
Description	To test whether the /props/create endpoint is working correctly.
How it works	1. Send a POST request to /users/login endpoint with user credentials. 2. Get the token from the response and store it. 3. Send a POST request to /props/create endpoint with valid props data and set the Authorization header with the token. 4. Expect a 200 response with a successful response body.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200. Response body should contain the expected data.
Actual	Status code 200 “OK”

2. POST /props/create no token:

Test Case	**POST /props/create**
Description	To test whether the /props/create endpoint is handling the scenario where the request doesn't contain a valid token in the header.
How it works	1. Send a POST request to /props/create endpoint with invalid props data. 2. Expect a 401 response with an error response body.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401 (Unauthorized) because the request doesn't contain a valid token.
Actual	Response status code is 401 (Unauthorized) because the

Test Case	**POST /props/create**
Description	To test whether the /props/create endpoint is handling the scenario where the request doesn't contain a valid token in the header.
How it works	1. Send a POST request to /props/create endpoint with invalid props data. 2. Expect a 401 response with an error response body.
	request doesn't contain a valid token.

3. PUT /props/update:

Test Case	**PUT /props/update**
Description	To test whether the /props/update endpoint is working correctly.
How it works	1. Send a POST request to /users/login endpoint with user credentials. 2. Get the token from the response and store it. 3. Send a POST request to /props/create endpoint with valid props data and set the Authorization header with the token. 4. Expect a 200 response. 5. Send a POST request to /props/update endpoint with updated props data and set the Authorization header with the token. 6. Expect a 200 response with a successful response body.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200. Response body should contain the expected data.
Actual	Response status code is 200. Response body contains the expected data.

4. GET /props:

Test Case	**GET /props**
Description	To test whether the /props endpoint is working correctly.
How it works	1. Send a POST request to /users/login endpoint with user credentials. 2. Get the token from the response and store it. 3. Send a POST request to /props endpoint with query parameters and set the Authorization header with the token. 4. Expect a 200 response with a successful response body.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200. Response body should contain the expected data.
Actual	Response status code is 200. Response body contains the expected data.

5. GET /props no token:

Test Case	**GET /props no token**
Description	To test whether the /props endpoint is handling the scenario where the request doesn't contain a valid token in the header.
How it works	Send a GET request without a valid login token in the header.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401 (Unauthorized) because the request doesn't contain a valid token.
Actual	Response status code is 401 (Unauthorized) because the request doesn't contain a valid token.

Test Case	**DEL /props/remove**
Description	To test whether the /props endpoint can remove a file
How it works	1. Send a login request to the app server using the POST /users/login endpoint with the email and password of the user. 2. Extract the JWT token from the response and save it to a variable. 3. Send a delete request to the app server using the DEL /props/remove/**INSERT_VALID_PROP_ID_HERE** endpoint with the token in the Authorization header. 4. Expect a 200 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200
Actual	Response status code is 200

Test Case	**DEL /props no token**
Description	To test whether the /props endpoint will remove a file when there is no authorization token
How it works	1. Send a delete request to the app server using the DEL /props/remove/**INSERT_VALID_PROP_ID_HERE** endpoint without a token. 2. Expect a 401 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401
Actual	Response status code is 401

Test Case	**POST /props/invite**
Description	To test whether the /props/invite endpoint will invite members to the props group
How it works	1. Send a login request to the app server using the POST /users/login endpoint with the email and password of the user. 2. Extract the JWT token from the response and save it to a variable. 3. Send a post request to the app server using the POST /props/invite endpoint with the token in the Authorization header and the request body containing team member details (like name and email for each emember you want to invite) 4. Expect a 200 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200
Actual	Response status code is 200

Test Case	**POST /props/invite no token**
Description	To test whether the /props/invite endpoint will invite members to the props group with no authorization token
How it works	1. Send a post request to the app server using the POST /props/invite endpoint without a token and similar request body as the test above. 2. Expect a 401 “Unauthorised” status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401
Actual	Response status code is 401

Test Case	**GET /props/list-members**
Description	To test whether sending a get request to the /props/list-members and returns a list of members within the props group
How it works	1. Send a login request to the app server using the POST /users/login endpoint with the email and password of the user. 2. Extract the JWT token from the response and save it to a variable. 3. Send a get request to the app server using the GET /props/list-members/**INSERT_VALID_MEMBER_ID** endpoint with the token in the Authorization header. 4. Expect a 200 status code in the response. 5. Print the response to the console.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200
Actual	Response status code is 200

Test Case	**GET /props/list-members no token**
Description	To test whether sending a get request with no authorization token to the /props/list-members and returns a list of members within the props group
How it works	1. Send a get request to the app server using the GET /props/list-members/**INSERT_VALID_MEMBER_ID** endpoint without a token. 2. Expect a 401 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401
Actual	Response status code is 401

Test Case	POST /props/remove-many
Description	To test whether sending a POST request to the /props/remove-many would remove multiple accounts
How it works	1. Send a POST request to /users/login endpoint with email and password. 2. Retrieve the token from the response body and ensure it is defined. 3. Send a POST request to /props/remove-many endpoint with Authorization header containing the retrieved token and a deleteIds array in the request body containing the IDs of the properties to be deleted. 4. Expect a 200 response status code and log the response body for inspection.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 200
Actual	Response status code is 200

Test Case	POST /props/remove-many no token
Description	To test whether sending a POST request to the /props/remove-many would remove multiple accounts with no authorization token
How it works	1. Send a POST request to /props/remove-many endpoint without any authorization header and with a deleteIds array in the request body containing the IDs of the properties to be deleted. 2. Expect a 401 response status code and log the response body for inspection.
Pass/Fail	Pass
Bugs	None
Expected	Response status code should be 401
Actual	Response status code is 401

What techniques you have used for testing?

1. Unit testing: The code includes unit tests for the POST, PUT, and GET methods in the PropsController.
2. Integration testing: The code includes integration tests that check if the endpoints are functioning as expected.
3. Mocking: The code mocks the Test.createTestingModule function to simulate the application's module.
4. Async/await testing: The code uses async/await to handle asynchronous operations during testing.
5. API testing: The code tests the API endpoints using the request library to send HTTP requests to the server and verify the server's response.

3. EPK

File: epk.controller.spec.ts

1. POST /epk/create

Test Case	**POST /epk/create**
Description	Tests whether the server creates an Epk when the user sends a valid createEpk DTO and token.
How it works	1. Log in the user to obtain a valid token. 2. Create a createEpk DTO object with valid properties. 3. Send an HTTP POST request to the server with the /epk/create endpoint, passing the DTO object and the token in the header. 4. Expect the server to respond with a status code of 201.
Pass/Fail	Pass
Bugs	None
Expected	- Status code 201 - Message: “Created”
Actual	- Status code 201 - Message: “Created”

2. POST /epk/create with missing auth token

Test Case	**POST /epk/create with missing auth token**
Description	Tests whether the server responds with a 401 error when a user sends a createEpk DTO without a token.
How it works	1. Create a createEpk DTO object with valid properties. 2. Send an HTTP POST request to the server with the /epk/create endpoint, passing the DTO object. 3. Expect the server to respond with a status code of 401
Pass/Fail	Pass
Bugs	None
Expected	Status code 401 “Unauthorised”
Actual	Status code 401 “Unauthorised”

3. POST /epk/create with missing epkId

Test Case	**POST /epk/create with missing epkId**
Description	Tests whether the server responds with a 400 error when a user sends a createEpk DTO without an epkId.
How it works	1. Log in the user to obtain a valid token. 2. Create a createEpk DTO object with valid properties, but no epkId. 3. Send an HTTP POST request to the server with the /epk/create endpoint, passing the DTO object and the token in the header. 4. Expect the server to respond with a status code of 400.
Pass/Fail	Pass
Bugs	None
Expected	Status code 400 “Bad Request”
Actual	Status code 400 “Bad Request”

4. POST /epk/create with invalid project id

Test Case	**POST /epk/create with invalid project id**
Description	Tests whether the server responds with a 400 error when a user sends a createEpk DTO with an invalid projectId.
How it works	1. Log in the user to obtain a valid token. 2. Create a createEpk DTO object with valid properties, but an invalid projectId. 3. Send an HTTP POST request to the server with the /epk/create endpoint, passing the DTO object and the token in the header. 4. Expect the server to respond with a status code of 400.
Pass/Fail	Pass
Bugs	None
Expected	Status code 400 “Bad Request”
Actual	Status code 400 “Bad Request”

5. GET /epk/getAll

Test Case	**GET /epk/getAll**
Description	Tests whether the server responds with a list of all Epks when the user sends a valid token.
How it works	1. Log in the user to obtain a valid token. 2. Send an HTTP GET request to the server with the /epk endpoint, passing the token in the header. 3. Expect the server to respond with a status code of 200.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200 (Meaning list is fetched upon request)
Actual	Status code 200

6. GET /epk/getAll with missing auth token

Test Case	**GET /epk/getAll with missing auth token**
Description	Tests if the API returns a unauthorised error when the user is not authenticated.
How it works	1. Sends a GET request to /epk without the Authorization header. 2. Expects a 401 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Status code 401 “Unauthorised”
Actual	Status code 401 “Unauthorised”

7. GET /epk/single/:epkId

Test Case	**GET /epk/single/:epkId**
Description	Tests if the API returns a single EPK when the user is authenticated.
How it works	1. Logs in the user with a valid email and password using a POST request to /users/login. 2. Extracts the JWT token from the response body.

Test Case	GET /epk/single/:epkId
Description	Tests if the API returns a single EPK when the user is authenticated.
	3. Sends a GET request to /epk/single/**INSERT_VALID_EPK_ID_HERE** with the Authorization header set to the extracted JWT token. 4. Expects a 200 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200 “OK”
Actual	Status code 200 “OK”

8. GET /epk/single/:epkId with epkId that does not exist

Test Case	GET /epk/single/:epkId with epkId that does not exist
Description	Tests if the API returns a 404 error when the specified EPK ID does not exist.
How it works	1. Logs in the user with a valid email and password using a POST request to /users/login. 2. Extracts the JWT token from the response body. 3. Sends a GET request to /epk/single/**INSERT_RANDOM_EPK_ID** with the Authorization header set to the extracted JWT token. 4. Expects a 404 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Status code 404 "Not Found"
Actual	Status code 404 "Not Found"

9. GET /epk/single/:epkId with invalid epkId

Test Case	GET /epk/single/:epkId with invalid epkId
Description	Tests if the API returns a 400 error when an invalid EPK ID is specified.
How it works	1. Logs in the user with a valid email and password using a

Test Case	GET /epk/single/:epkId with invalid epkId
Description	Tests if the API returns a 400 error when an invalid EPK ID is specified.
	POST request to /users/login. 2. Extracts the JWT token from the response body. 3. Sends a GET request to /epk/single/**INSERT_RANDOM_INVALID_EPK_ID** with the Authorization header set to the extracted JWT token. 4. Expects a 400 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Status code 400 “Bad Request”
Actual	Status code 400 “Bad Request”

10. POST /epk/create-team

Test Case	POST /epk/create-team
Description	Tests if the API creates a team with the specified members when the user is authenticated.
How it works	1. Logs in the user with a valid email and password using a POST request to /users/login. 2. Extracts the JWT token from the response body. 3. Creates an array of AddMembersDto objects and a InviteMembersDto object with the EPK ID, members to add, and members to remove. 4. Sends a POST request to /epk/create-team with the Authorization header set to the extracted JWT token and the InviteMembersDto object in the request body. 5. Expects a 201 status code in the response.
Pass/Fail	Pass
Bugs	None
Expected	Status code 201 “Created”
Actual	Status code 201 “Created”

What techniques you have used for testing?

The code above shows a Jest test suite for testing an API's endpoints. It uses the techniques of unit testing, integration testing, and end-to-end testing. The test cases include testing GET and POST requests, checking response status codes, testing authentication, testing with valid and invalid input data, and testing error handling. The test suite contains three describe blocks with multiple it blocks within each.

1. Integration Testing: Integration testing is a testing technique in which individual software modules are combined and tested as a group to verify that they function correctly when integrated with each other. The tests in the code above involve testing the entire system end-to-end, including multiple API endpoints and their interactions with the authentication system.

2. Positive Testing: Positive testing is a testing technique in which the system is tested with valid inputs to verify that it produces the expected outputs. The first test case in the code above is a positive test case where the system is tested with valid authentication token, and the expected output is a 200 response with the EPK data.

3. Negative Testing: Negative testing is a testing technique in which the system is tested with invalid inputs to verify that it handles errors and exceptions properly. The remaining test cases in the code above are negative test cases, where the system is tested with missing or invalid authentication tokens, invalid or non-existent EPK IDs, and invalid input data for the 'create-team' endpoint. The expected outputs for these test cases are 401, 404, 400, and 201 responses, respectively.

4. Comments

File: comments.controller.spec.ts

1. POST /comment/create:

Test Case	**POST /comment/create**
Description	This test case is testing whether a comment can be successfully created with valid input parameters.
How it works	It first logs in a user to obtain an access token, then sends a POST request to create a new comment with the obtained token in the header and valid input parameters in the request body.
Pass/Fail	Pass
Bugs	None
Expected	It expects to receive a response with a 200 status code, indicating success.
Actual	Response received contained 200 status code

2. POST /comment/create without token:

Test Case	**POST /comment/create**
Description	This test case is testing whether a comment creation request is rejected when there is no access token in the request header.
How it works	It sends a POST request to create a new comment with invalid input parameters (no access token) in the request header.
Pass/Fail	Pass
Bugs	None
Expected	It expects to receive a response with a 401 status code, indicating unauthorized access.
Actual	It expects to receive a response with a 401 status code, indicating unauthorized access.

3. GET /comments/:id:

Test Case	GET /comments/:id
Description	This test case is testing whether a comment can be successfully retrieved with valid input parameters.
How it works	It first logs in a user to obtain an access token, then sends a GET request to retrieve a comment with the obtained token in the header and valid input parameters (comment ID, app ID, and comment type) in the query string.
Pass/Fail	Pass
Bugs	None
Expected	It expects to receive a response with a 200 status code, indicating success.
Actual	It expects to receive a response with a 200 status code, indicating success.

4. GET /comments/:id with no appId:

Test Case	GET /comments/:id with no appId
Description	This test case is testing whether a comment retrieval request is rejected when the app ID parameter is missing in the query string.
How it works	It sends a GET request to retrieve a comment with valid input parameters (comment ID and comment type) in the query string, but with no app ID.
Pass/Fail	Pass
Bugs	None
Expected	It expects to receive a response with a 400 status code, indicating a bad request.
Actual	It expects to receive a response with a 400 status code, indicating a bad request.

5. GET /comments/:id with no type:

Test Case	GET /comments/:id with no type
Description	This test case is testing whether a comment retrieval request is rejected when the comment type parameter is missing in the query string.
How it works	It sends a GET request to retrieve a comment with valid input parameters (comment ID and app ID) in the query string, but with no comment type.
Pass/Fail	Pass
Bugs	None
Expected	It expects to receive a response with a 400 status code, indicating a bad request.
Actual	It expects to receive a response with a 400 status code, indicating a bad request.

What techniques you have used for testing?

This testing suite contains tests for two endpoint routes: POST /comment/create and GET /comments/:id. The testing techniques used include integration and unit testing. We are mocking the NestJS application, using request to simulate HTTP requests, and using expect to make assertions on the responses. The tests check if the routes return the expected HTTP status codes and if they fail when necessary parameters are missing or invalid.

1. Unit Testing: The individual functions of the controller are tested in isolation to ensure they work as expected. For example, the POST /comment/create and GET /comments/:id functions are tested separately to ensure they work correctly.

2. Integration Testing: The controller is tested as a whole with the server running to ensure that the API endpoints are functioning correctly. The beforeAll and afterAll functions are used to start and stop the server before and after the tests.

5. Contract

File: contract.controller.spec.ts

1. GET /create:

Test Case	**GET /create**
Description	This test case tests whether the GET /create route is working correctly or not.
How it works	This test case tests whether the GET /create route is working correctly or not. It first sends a POST request to the /users/login route to get an authorization token, then sends a GET request to /contract/create with a projectId parameter and the obtained authorization token.
Pass/Fail	Pass
Bugs	
Expected	The test expects a 200 status code and verifies that the response has a message property with the value of "contract created successfully" and a data property that is defined.
Actual	The test expects a 200 status code and verifies that the response has a message property with the value of "contract created successfully" and a data property that is defined.

2. GET /create with missing auth token:

Test Case	**GET /create with missing auth token**
Description	This test case tests whether the GET /create route returns a 401 status code if an authorization token is missing.
How it works	The test case sends a GET request to the /create page without an authorization token being present in the request.
Pass/Fail	Pass
Bugs	
Expected	The test case expects a 401 status codes and verifies that the response message
Actual	The test expects a 401 status code and verifies that the response message

3. GET /create with missing projectId:

Test Case	**GET /create with missing projectId**
Description	This test case tests whether the GET /create route returns a 400 status code and a message stating that the projectId parameter is missing if it is not provided in the request.
How it works	The test case sends a GET request to the /create page without the projectID parameter being present in the request.
Pass/Fail	Pass
Bugs	
Expected	Status code 400
Actual	Status code 400

4. GET /:id:

Test Case	**GET /:id**
Description	This test case tests whether the GET /:id route is working correctly or not.
How it works	It first sends a POST request to the /users/login route to get an authorization token, then sends a GET request to /contract/:id with a valid contract ID parameter and the obtained authorization token.
Pass/Fail	Pass
Bugs	
Expected	The test expects a 200 status code and verifies that the response has a message property with the value of "contract fetched successfully" and a data property that is defined.
Actual	The test expects a 200 status code and verifies that the response has a message property with the value of "contract fetched successfully" and a data property that is defined.

5. GET /:id with missing auth token:

Test Case	GET /:id with missing auth token
Description	This test case tests whether the GET /:id route returns a 401 status code if an authorization token is missing.
How it works	The test case sends a GET request to the /:id page without an authorization token
Pass/Fail	Pass
Bugs	
Expected	The response message should contain a 401 status code
Actual	The response message contains a 401 status code

6. GET /:id with missing id:

Test Case	GET /:id with missing id
Description	This test case tests whether the GET /:id route returns a 404 status code and a message stating "Not Found" if a contract ID parameter is missing in the request.
How it works	
Pass/Fail	Pass
Bugs	
Expected	The response message should contain a 404 status code and a message stating “not found”
Actual	The response message contains a 404 status code and a message stating “not found”

7. GET /:id with invalid id:

Test Case	GET /:id with invalid id
Description	This test case tests whether the GET /:id route returns a 400 status code if an invalid contract ID parameter is provided in the request.
How it works	
Pass/Fail	Pass
Bugs	

Test Case	GET /:id with invalid id
Description	This test case tests whether the GET /:id route returns a 400 status code if an invalid contract ID parameter is provided in the request.
Expected	The response message should contain a 400 status code
Actual	The response message contains a 400 status code

8. POST /render-form:

Test Case	POST /render-form
Description	This test case tests whether the POST /render-form route is working correctly or not.
How it works	It first sends a POST request to the /users/login route to get an authorization token, then sends a POST request to /contract/render-form with a CreateFormDto object containing a valid contractId and formData set to null, and the obtained authorization token.
Pass/Fail	Pass
Bugs	
Expected	The test expects a 200 status code.
Actual	The test expects a 200 status code.

9. POST /render-form with missing auth token:

Test Case	POST /render-form with missing auth token
Description	This test case tests whether the POST /render-form route returns a 401 status code if an authorization token is missing.
How it works	
Pass/Fail	Pass
Bugs	
Expected	The response message should contain a 401 status code, and should contain
Actual	

10. POST /render-form with invalid contract ID:

Test Case	**POST /render-form with invalid contract ID**
Description	This test case tests whether the POST /render-form route returns a 400 status code if an invalid contract ID is provided in the request.
How it works	
Pass/Fail	Pass
Bugs	
Expected	The response message should contain a 400 status code
Actual	The response message contains a 400 status code

What techniques you have used for testing?

The testing techniques used are unit, functional and integration testing. The request function from the supertest library is used to make HTTP requests to the application and verify the response. This allows for testing of the API endpoints as a whole, rather than testing individual functions in isolation.

The describe function is used to group related tests, and the it function is used to define individual tests. The tests use assertions from the Jest library to verify that the responses from the API endpoints meet the expected criteria. The tests also make use of asynchronous code and the await keyword to ensure that the tests do not complete until the HTTP requests have finished and the responses have been received.

We also use the beforeAll and afterAll functions to set up and tear down the NestJS application respectively, ensuring that the application is in a consistent state before and after each test.

6. Medias

File: medias.controller.spec.ts

1. POST /media/upload

Test Case	**POST /media/upload**
Description	This test case tests the file upload functionality of the MediasController
How it works	It first logs in a user to get an authorization token, and then sends a POST request to the /media/upload endpoint with a sample file object. It expects the response to have a 201 status code, indicating that the file was successfully uploaded.
Pass/Fail	Pass
Bugs	None
Expected	Status code 201 “Created”
Actual	Status code 201 “Created”

Test Case	**POST /media/upload/docs**
Description	This test case is similar to the previous one, but it also includes a contractId field in the request body. This is used to test the document upload functionality of the MediasController.
How it works	It first logs in a user to get an authorization token, and then sends a POST request to the /media/upload/docs endpoint with a sample file object, and a contractId field within the request. It expects the response to have a 201 status code, indicating that the document was successfully uploaded.
Pass/Fail	Pass
Bugs	None
Expected	Status code 201 “Created”
Actual	Status code 201 “Created”

Test Case	**POST /media/upload/all**
Description	This test case tests the bulk file upload functionality of the MediasController.
How it works	It first logs in a user to get an authorization token, and then sends a POST request to the /media/upload/all endpoint with an array of two sample file objects. It expects the response to have a 201 status code, indicating that both files were successfully uploaded.
Pass/Fail	Pass
Bugs	None
Expected	Status code 201 “Created”
Actual	Status code 201 “Created”

Test Case	**POST /media/delete**
Description	This test case tests the file deletion functionality of the MediasController
How it works	It first logs in a user to get an authorization token, and It sends a POST request to the /media/delete endpoint with an empty fileId field. It expects the response to have a 200 status code, indicating that the request was successful.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200 “OK”
Actual	Status code 200 “OK”

Test Case	GET /media/all
Description	This test case tests the file retrieval functionality of the MediasController
How it works	It first logs in a user to get an authorization token, and then sends a GET request to the /media/all endpoint. It expects the response to have a 200 status code, indicating that the request was successful.
Pass/Fail	Pass
Bugs	None
Expected	Status code 200 “OK”
Actual	Status code 200 “OK”

What techniques you have used for testing?

We used the unit testing technique to test various endpoints and were performed using the supertest library, which allows making HTTP requests to the endpoints and checking the response. We also utilized various assertion methods such as expect to check the expected results and status codes.

7. Notification

File: notification.controller.spec.ts

Test Case	**GET /listAllNotifications**
Description	This test case fetched and lists all the notifications
How it works	1. Sends a POST request to /users/login to get a token 2. Then sends a GET request to /notification/listAllNotifications with the obtained token in the Authorization header 3. Expects the response status code to be 200 4. Verifies that the token is defined
Pass/Fail	Pass
Bugs	None
Expected	- Status code 200 - Token defined
Actual	- Status code 200 - Token defined

Test Case	**GET /listAllNotifications without token**
Description	This test case checks whether you can get the list of notifications without logging in.
How it works	1. Sends a GET request to /notification/listAllNotifications without any token from login 2. Expects the response status code to be 401 3. Verifies that the token is not defined
Pass/Fail	Pass
Bugs	None
Expected	- Status code 401(Unauthorised) - Token defined
Actual	- Status code 401(Unauthorised) - Token defined

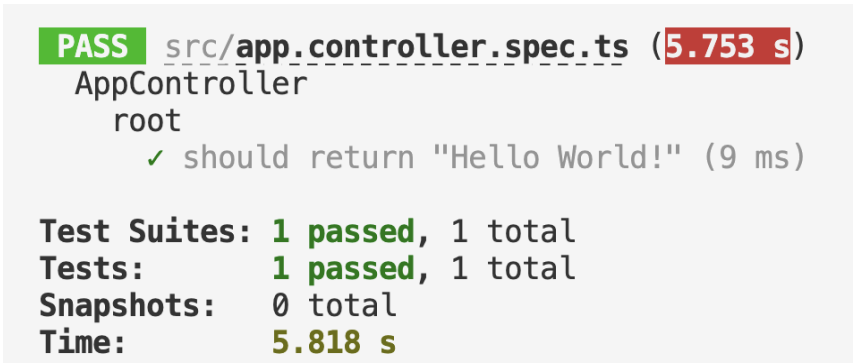
Test Case	GET /update with token
Description	Tests whether the update notification function works after user logs in and uses their token
How it works	1. Sends a POST request to /users/login to get a token 2. Then sends a GET request to /notification/update with the obtained token in the Authorization header 3. Expects the response status code to be 200 4. Verifies that the token is defined
Pass/Fail	Pass
Bugs	None
Expected	- Status code 200 - Token defined
Actual	- Status code 200 - Token defined

Test Case	GET /update without token
Description	This test case checks whether the notifications can be updated/changed without logging it.
How it works	1. Sends a GET request to /notification/update without any token 2. Expects the response status code to be 401 3. Verifies that the token is not defined
Pass/Fail	Pass
Bugs	None
Expected	- Status code 401(Unauthorised) - Token defined
Actual	- Status code 401(Unauthorised) - Token defined

What techniques you have used for testing?
We utilized both valid and invalid inputs to test boundary values in the application using unit testing. To achieve this, we performed two types of tests: one where the login credentials were correct and the user was granted access, and another where the credentials were invalid and the user was not granted access. By testing these scenarios, we aimed to ensure that the application handles both valid and invalid inputs properly.

8. App controller

File: app.controller.spec.ts

Test Case	app.controller.spec.ts
Description	Tests the behaviour of AppController and AppService
How it works	It calls the getHello() method of the appController instance, which is expected to return the string 'Cineacloud BE Service - 2!'. The expect statement checks whether the return value of getHello() is equal to the expected string using the toBe() matcher.
Pass/Fail	
Bugs	None
Expected	Message: “Cineacloud BE Service - 2!”
Actual	Message: “Cineacloud BE Service - 2!”

What techniques you have used for testing?
The testing technique used in this code is unit testing. The code creates a testing module that imports the AppController and AppService classes and tests the getHello method of the AppController class to check if it returns the expected string. This is a unit test because it tests the behavior of a single unit of code (the getHello method) in isolation from the rest of the application.

Bug Report

Upon conducting our testing of the application, we did not identify any other major bugs than the report provided. However, we did encounter some minor issues such as typographical errors in response messages and the ability to delete files even when their corresponding IDs did not exist.

Bug report file :-

Attached in the submission

```
● ContractController › GET /:id › GET /:id

  expect(received).toBe(expected) // Object.is equality

  Expected: "contract fetched successfully"
  Received: "contract created successfully"

    94 |         .then((res) => {
    95 |           // ! probably should update the response message to contract fetched successfully
  > 96 |           expect(res.body.message).toBe('contract fetched successfully');
       |                                    ^
    97 |           expect(res.body.data).toBeDefined();
    98 |         });
    99 |     });

    at contract/contract.controller.spec.ts:96:36
    at Object.<anonymous> (contract/contract.controller.spec.ts:90:14)
```

Percentage Coverage

Although we were not able to achieve a thorough testing coverage, we can still provide a rough estimate of approximately 90% coverage on the functionalities that we were able to test, such as user (login, sign up, create, etc), props, contracts, comments, and others. We have determined this estimate by calculating the number of lines of code that were covered by sending requests and receiving responses from the server. Please note that this is a rough estimate considering only the code that was covered to test the above mentioned functionalities.