Software Quality
Ola 2
API Testing, Coverage, and Benchmarking

By Kristofer, Mads, Michael and Søren

Create a Simple REST API:

We continued to develop on our application we made for our last assignment, a to-do task board, that consists of a frontend, a backend and a database. We didn't refactor any of the existing code for the application as we felt it already met the requirements stated in the work assignment document, with the exception of one bug fix. We did however add new code in the form of more tests.

Unit and Integration Testing:

Unit Tests:

We didn't feel the need to add any new unit tests or change our already existing ones, as we believe they already meet the requirements stated in the work assignment document. We will, for clarification of what unit test we have in our project, write them down again.

- Test 1 - Adding a to-do task. - Test API

Function	Successful result
AddTask creates a mock task and then attempts to create a new task containing the same text.	The mock task contains the same values as the created task.

- Test 2 - Deleting a to-do task. - Test API

Function	Successful result
DeleteTask sends along an ID of the task that is meant to be deleted, finds the task at hand and continues to delete it.	When the selected task with the id sent is deleted.

- Test 3 - Edit to-do task text. - Test API

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

- Test 4 - Edit to-do task is complete state. - Test Function

Function	Successful result
EditCompleteState sends an ID of the task that is desired to be edited and the current "completion state" of the task. Based on this, the state is updated to the opposite on the task with the matching ID.	If the completion state of the targeted task is set to the opposite of the current state.

- Test 5 - Edit to-do task list position (Category ID). - Test Function

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

- Test 6 - Edit to-do task deadline. - Test Function

Function	Successful result
EditTask creates a mock task and then first attempt to update an already existing task with the same id to match all values	The mock task contains the same values as the updated task.

Integration Tests:

We again have tests from our previous assignment but here we have created new integration tests that aim to test our simulated API calls. We have marked the new tests we made a green color to show which are new.

- Test 1 - Updating to-do tasks text:

Function	Successful result
EditTask creates a dummy task and then first attempt to update an already existing task with the same id to match all values	The mock task was successfully updated in the db and contains the same values as the updated task.

- Test 2 -adding task:

Function	Successful result
AddTask creates a mock task and then attempts to create a new task containing the same text.	The mock task was successfully created in the database and contains the same values as the created task.

- Test 3 - API - Get “/tasks”:

Function	Successful result
Create a result from an asynchronous get request to the api get endpoint “/tasks”.	Result statusCode equals 200

- Test 4 - API - Put “/tasks”:

Function	Successful result
Create a result from an asynchronous put request to the api put endpoint “/tasks/{id}” using a dummy task and its id.	Result statusCode equals 200

- Test 5 - API - Post “/tasks/:id”:

Function	Successful result
Create a result from an asynchronous get request to the api post endpoint “/tasks” using a dummy task.	Result statusCode equals 200

Code Coverage:

We used Jest’s inbuilt feature to check our code coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	88.88	55.55	88.88	87.61	
src	83.07	33.33	84.61	81.66	
index.ts	80.35	0	84.61	78.43	28-29, 41-42, 68-70, 82-84, 90-96
ormconfig.ts	100	50	100	100	9
src/db_functions	95.34	66.66	100	94.73	
taskRepository.ts	95.34	66.66	100	94.73	54, 67
src/entities	100	100	100	100	
Task.ts	100	100	100	100	

We have a very good code coverage of 88.9%, this is because we have tests that cover all of our crud operations. The test report shows we have extensive code coverage, except for some lines in index.ts. When we look at the lines of code, these are catches or if statements for error handling. This makes sense as we have many tests that test for successes but not for errors or status codes other than 200.

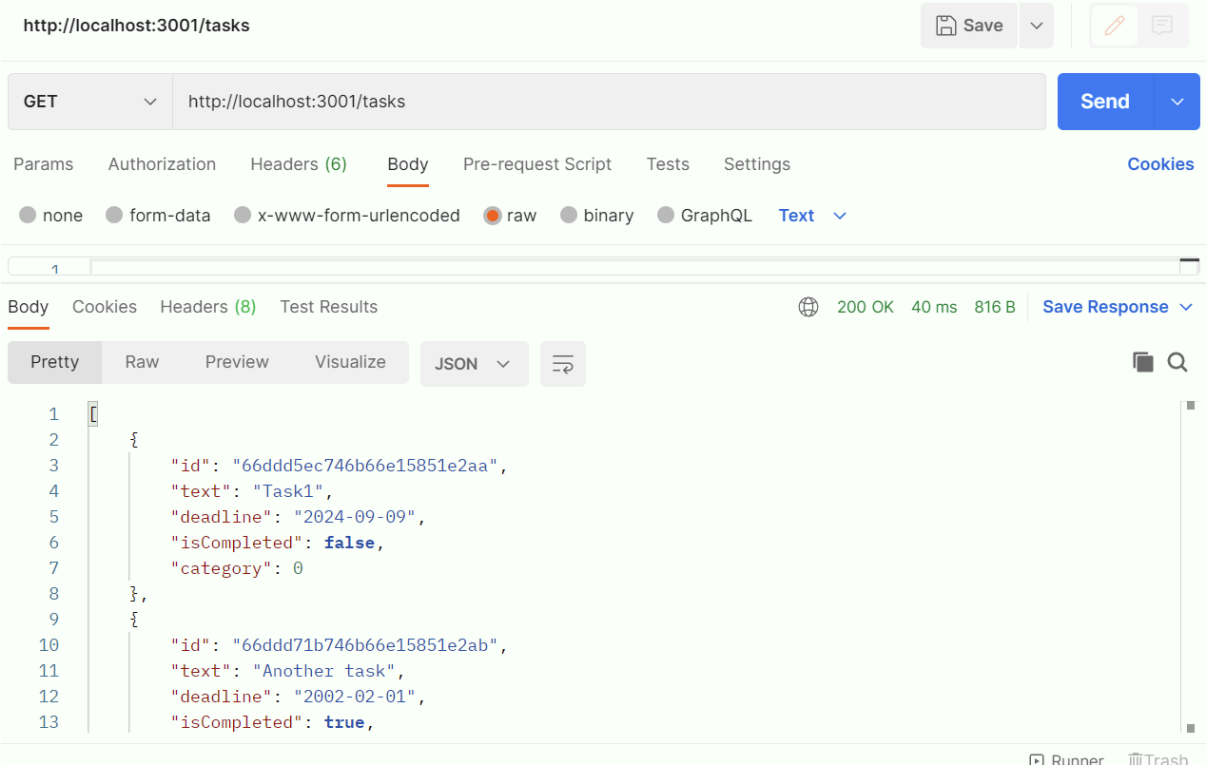
Reflection on Coverage and Performance:

Testing for success and not failure:

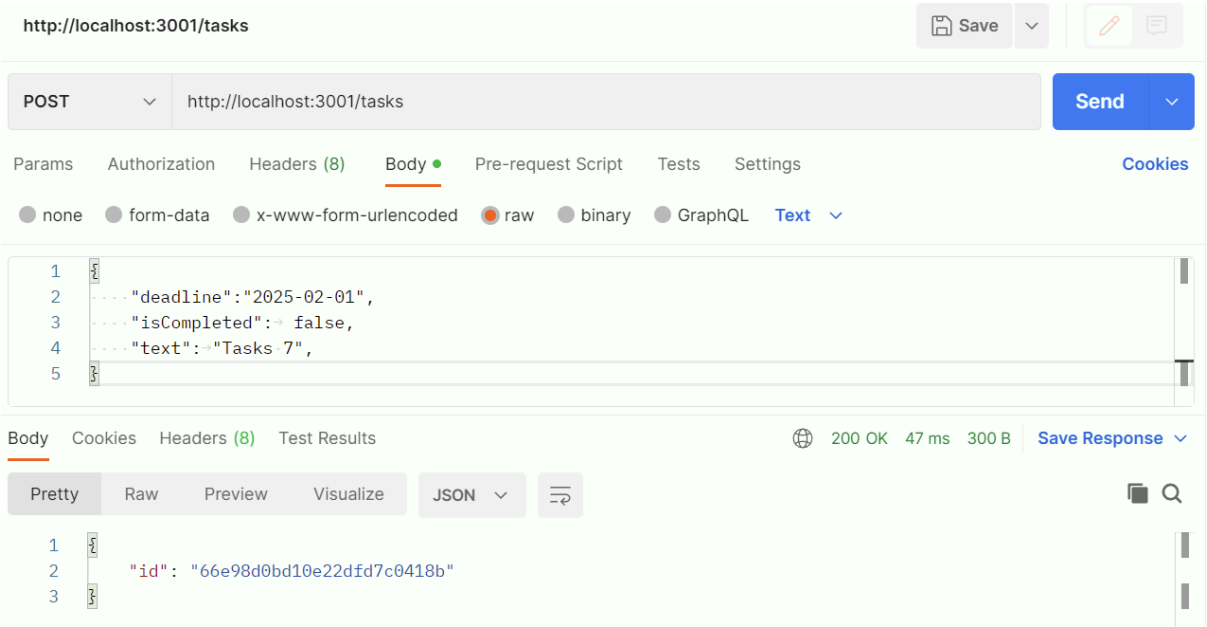
Going over all of our tests, we quickly see that all of our tests, be they unit tests, integration tests or even specification tests, all test with the focus on achieving an expected result or “Testing for success”. While the tests do work, it means we have a lack of coverage of what we can expect to fail. An example on making an integration test for our api that focuses on a failed result, would be to send an id for a task in the negative like “-1”.

Postman API testing:

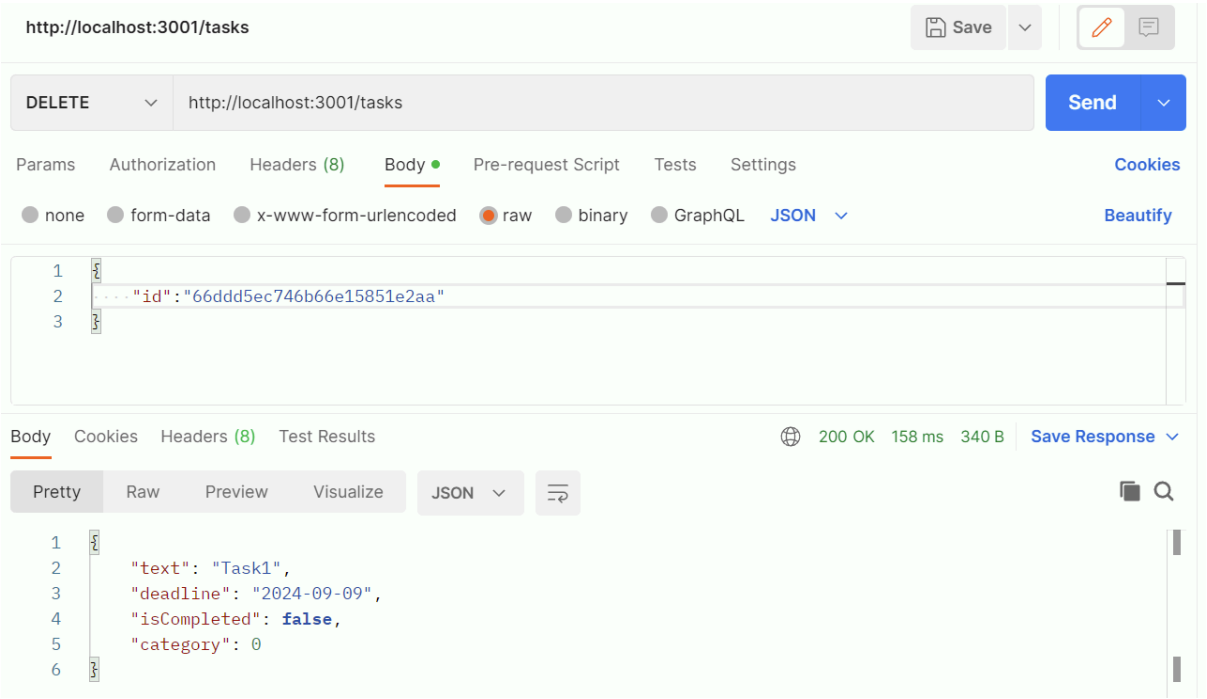
Get:
This GET api receives all the tasks in the database.



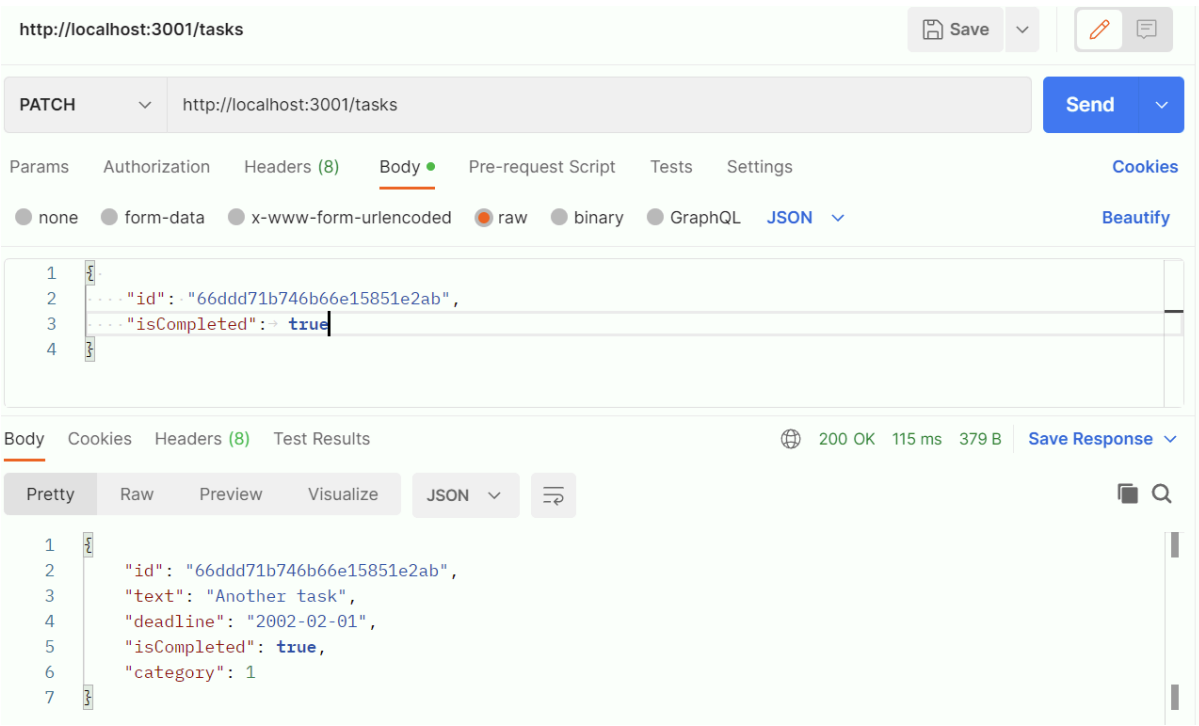
Post:
This POST api creates a new task. A “text”, which is basically the title, a “deadline” and a “isCompleted” can all be parsed to the api, but only the “text” and “isCompleted” is required. The api returns the API of the newly created task.



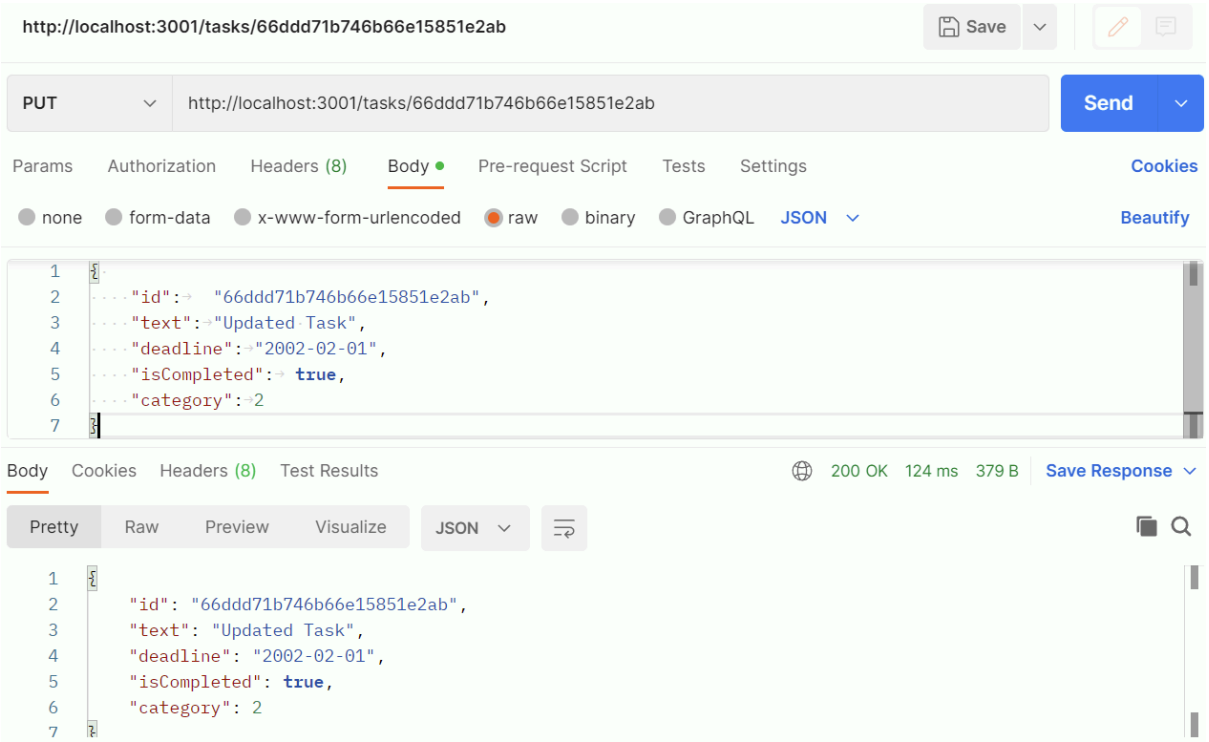
Delete:
This DELETE api deletes a task. The body only needs to take the id of the task that is supposed to be deleted. The task that got deleted is then returned.



Complete:
This PATCH api updates a task to mark it as either complete or incomplete, depending on the boolean “isCompleted”. The task with the new complete state is then returned.



Edit Task:
This PUT api updates the values of a task, such as the category, text and deadline. The task with the changes implemented is then returned.



Basic Load Testing with JMeter (or similar tool):

Test Details:

- **Test Run ID:** tjaj3_wt4553nawjrcfkpewk3cankc4rd35_e3zb
- **Phase Started:** Ramp-up to 75 users per second
- **Total Duration:** 1 minute, 2 seconds
- **Total Requests:** 4500
- **Virtual Users (VUs) Created:** 4500
- **Virtual Users (VUs) Completed:** 4500
- **Virtual Users Failed:** 0

Key Metrics Overview

1. **Throughput:**
    - **Requests per second:** 75/sec
    - **Total Requests:** 4500
    - **HTTP 200 Responses:** 4500
    - **Downloaded Data:** 1,642,500 bytes (1.57 MB)
2. **Response Times:**
    - **Minimum Response Time:** 17 ms
    - **Maximum Response Time:** 667 ms
    - **Mean Response Time:** 27.8 ms
    - **Median Response Time:** 22 ms
    - **95th Percentile (P95):** 50.9 ms
    - **99th Percentile (P99):** 122.7 ms
3. **Session Length (Time VUs Spent Executing):**
    - **Minimum Session Length:** 18.7 ms
    - **Maximum Session Length:** 667.8 ms
    - **Mean Session Length:** 29.7 ms
    - **Median Session Length:** 23.8 ms
    - **95th Percentile (P95):** 53 ms
    - **99th Percentile (P99):** 125.2 ms

Analysis

1. **Throughput and Stability:**
    - The system maintained a steady throughput of 75 requests per second, with consistent 200 HTTP response codes across all requests.
    - No virtual users (VUs) failed during the test, indicating the API handled the load without server errors or failures.
2. **Response Time Distribution:**
    - **Mean Response Time:** The average response time remained at 27.8 ms, which is well within a favorable range for API performance under load.
    - **Median Response Time:** At 22 ms, most requests were processed very quickly, indicating efficient handling of typical user requests.

    - **P95 and P99 Response Times:** The response times for the 95th and 99th percentiles are 50.9 ms and 122.7 ms, respectively. The higher P99 time suggests that while most responses were fast, a small number of requests (top 1%) took longer to process, possibly due to heavier resource use or network delays.
3. **Session Length Distribution:**
    - **Mean Session Length:** At 29.7 ms, this is consistent with the mean response time, showing that virtual users generally completed their tasks quickly.
    - **Maximum Session Length:** Some outliers took up to 667.8 ms. This is likely due to the tail end of users experiencing slower-than-average network or system conditions.

Conclusion:

- **API Stability:** The API performed well under moderate load (75 requests per second), with no failures, stable throughput, and consistent 200 OK responses.
- **Performance:** Response times were excellent, with a median of 22 ms and a P95 of 50.9 ms, indicating the API handled most requests efficiently. The longer response times (P99: 122.7 ms) were infrequent and within acceptable limits for most real-time systems.
- **Potential Areas for Focus:** While the majority of the requests were processed efficiently, monitoring the slower response times (in the 99th percentile) may be useful to ensure consistent performance under higher load.

Overall, the API demonstrated good performance and can handle the moderate load efficiently. Future tests under higher load could help identify performance bottlenecks.

Because our application is built to only run locally between the frontend and the backend, means we also have set up our load test to function locally. This does affect our results in a big way as instead of testing if a server can handle the amount of request, between one backend and many frontends, we instead test only if the computer itself can process them fast enough. There is no data transferred over the internet and no data that is sent back over the internet. This also means we can expect a difference between the resulting data based on how good the computer is. Because of this, the value of our load test as part of our project, isn't worth much, but as a project the project having the testing created could later on be changed to test our backend when/if it is deployed as a cloud application.