

Assignment 1 - Cloud Based Web Services - Serena Jonathan - 2306363

by Serena Jonathan

Submission date: 28-Nov-2025 04:58AM (UTC+0000)

Submission ID: 268941181

File name: 50568_Serena_Jonathan_Assignment_1_-_Cloud_Based_Web_Services_-_Serena_Jonathan_-_2306363_708807_1301347089.docx (1.15M)

Word count: 3598

Character count: 21805



ACCA6024: CLOUD BASED WEB SERVICES

Abstract

This project provides a concise demonstration of software testing for a simple to-do list application built around basic CRUD operations. A series of structured test tables has been created to outline the purpose, method, and expected outcome of each test. These tests cover the core create, read, update, and delete functions to ensure that tasks within the to-do list can be added, viewed, edited, and removed as intended. The testing approach focuses on clarity, consistency, and accurate verification of results. Overall, this ensures that the to-do list system operates reliably and meets the required functional standards.

Serena Jonathan (2306363)

2306363@student.uwtsd.ac.uk

Contents

Introduction	2
Task 1	2
What is REST?	2
Key Concepts of REST APIs	2
Stateless:	2
Client-server:	3
Resources:	3
HTTP methods (GET, POST, PUT, PATCH, DELETE):	3
Advantages of REST APIs	3
Disadvantages of REST APIs	3
Alternative 1: GraphQL	4
Advantages of GraphQL	4
Disadvantages of GraphQL	4
Alternative 2: gRPC	4
Advantages of gRPC	5
Disadvantages of gRPC	5
Comparison: REST vs GraphQL vs gRPC	5
Use-cases	6
Conclusion	7
Bibliography	Error! Bookmark not defined.

Introduction

This assignment will explore the concept of REST APIs in detail and compares them with two alternative protocols used for accessing and exchanging information. The discussion will highlight the advantages, disadvantages, and appropriate use-cases for each approach. In addition to the theoretical evaluation, the assignment also includes the design, development, testing, and deployment of a working web API within a cloud environment, demonstrating core CRUD operations, workflow processing, and software testing practices. The following sections begin by examining the fundamental principles that define REST and how they shape modern API design.

Task 1

What is REST?

APIs or "Application Programming Interfaces," are software tools that enable different computer programs to communicate with one another. They function as a bridge between applications, allowing them to exchange information and work together. APIs are increasingly becoming a major part of modern business. [1] Specifically, Rest APIs are a type of API that follow specific rules and architecture principles defined by REST. REST stands for Representational State Transfer. [2] It is the protocol used for web APIs as they optimise client-server interactions.

Key Concepts of REST APIs

Some key concepts of REST APIs include stateless, client-server, resources, HTTP methods (GET, POST, PUT/PATCH and DELETE.) [3] Understanding these key concepts is essential for designing efficient, scalable, and maintainable web APIs. An overview of each concept is presented below.

Stateless:

A REST API is stateless, meaning that each client request contains all the information required for the server to process it, with no reliance on stored session data. [4]

Client-server:

REST follows a client-server architecture, separating the user interface and client logic from the server's data management and processing responsibilities. [5]

Resources:

In REST, data and functionality are represented as resources, each of which is uniquely identified using a URL. [1]

HTTP methods (GET, POST, PUT, PATCH, DELETE):

REST APIs use standard HTTP methods to manipulate resources: GET retrieves data, POST creates new resources, PUT replaces existing resources, PATCH updates part of a resource, and DELETE removes a resource. [6]

Advantages of REST APIs

REST APIs are widely recognised for their simplicity and broad adoption across web development and cloud-hosted applications. Their design adheres to standard HTTP methods, such as GET, POST, PUT, PATCH, and DELETE, which provides a consistent and predictable interface for developers. [7] This standardisation facilitates rapid integration with diverse clients and platforms, making REST a reliable choice for building cross-platform compatible systems. Additionally, REST APIs scale effectively due to their stateless nature, allowing servers to handle multiple requests independently and efficiently, which is particularly advantageous in high-traffic or distributed cloud environments. These features collectively contribute to the enduring popularity and practicality of REST APIs in modern software development. [8]

Disadvantages of REST APIs

Despite their widespread use and simplicity, REST APIs also have certain disadvantages. For example, they can be less efficient when handling complex operations that require multiple related resources, often requiring several separate requests to retrieve all the required data. This can result in slower response times and increased data transfer overhead, particularly in applications with intricate relationships between resources. [9]

Furthermore, REST does not provide a standardised approach to error handling; the format and content of error responses generally depend on the individual implementation, which can lead to inconsistencies across different APIs. Consequently, while REST is highly practical for many scenarios, it may not be the optimal choice for applications requiring highly complex queries or consistent error reporting. [7]

Alternative 1: GraphQL

GraphQL is a query language and runtime for APIs, originally developed by Facebook in 2012 and released publicly in 2015. [10] Unlike REST, which relies on multiple endpoints for different resources, GraphQL exposes a single endpoint that enables clients to specify exactly which data they require. [11] This precise control over queries provides flexibility and can reduce both over-fetching and under-fetching of data, which is a common limitation in traditional REST APIs. At its core, GraphQL is strongly typed, which means the API's schema specifies the structure of the data that clients can query. This allows clients to understand precisely which queries are valid and what data will be returned. [12] Furthermore, the single-endpoint approach simplifies client-side logic, as developers do not need to manage multiple URLs for various resources. [10]

Advantages of GraphQL

One of the main advantages of GraphQL is its efficiency in data retrieval. [13] Clients can request only the fields they require, thereby reducing unnecessary data transfer and improving performance, particularly in mobile applications where bandwidth may be constrained. The strongly typed schema also enhances reliability, as clients can identify errors at the query construction stage rather than at runtime. [14] Additionally, GraphQL is well suited to complex or nested data structures, allowing multiple related resources to be retrieved in a single request rather than issuing multiple REST calls to the server to get all the information you need. [15]

Disadvantages of GraphQL

Despite its advantages, GraphQL has its limitations. For example, it can be more complex to implement than REST, as the server requires a robust schema definition and query resolution logic. [16] Caching is also more challenging because responses are determined dynamically by the query rather than by fixed endpoints. Furthermore, the learning curve for developers unfamiliar with GraphQL can be steep, particularly when integrating it with existing REST infrastructure. [15]

Alternative 2: gRPC

gRPC is a high-performance, open-source communication protocol developed by Google. It uses HTTP/2 as its transport layer and relies on Protocol Buffers (often called protobufs) to define data structures and serialise messages. [17] Unlike REST, gRPC is designed for

fast, efficient communication between services, making it particularly suitable for large distributed systems and microservice architectures responsible for each specific task. [18]

Advantages of gRPC

One advantage of gRPC is that it has a superior performance. As gRPC uses binary serialisation rather than text-based formats like JSON, message sizes are significantly smaller. This reduces network usage and improves speed. [19] In addition to this, gRPC allows multiplexing. This means multiple requests can be sent over simultaneously on the same connection without interference. [20] Furthermore, gRPC has a strongly typed contract which ensures both sides follow the same data format. This reduces ambiguity, prevents type-related errors, and guarantees consistent communication across services. Lastly, its support for bidirectional streaming makes it well suited to real-time applications such as live analytics, chat systems, or IoT monitoring. [21]

Disadvantages of gRPC

Despite its advantages, gRPC has various disadvantages. For example, it is less suited to public-facing web APIs because most browsers do not natively support HTTP/2 streaming or Protocol Buffers, meaning additional tools or proxies are required for gRPC to work effectively. [22] Additionally, gRPC's binary messages are not readable in plain text, this makes debugging more difficult compared with REST. Although the protocol for gRPC is powerful, it also has a steeper learning curve and may be unnecessarily complex for simpler applications that do not require high-performance communication. Finally, because Protocol Buffers and service definitions must be compiled, development is less flexible than REST's more loosely structured, JSON-based approach. [23]

Comparison: REST vs GraphQL vs gRPC

REST, GraphQL, and gRPC each present different strengths and weaknesses, and the most suitable protocol depends heavily on the requirements of the system. REST remains the most widely adopted due to its simplicity, readability, and compatibility with standard HTTP tools. Its use of multiple, fixed endpoints makes caching straightforward and predictable. However, REST can become inefficient when dealing with highly relational or nested data, as multiple calls may be required to gather all needed information. [24]

On the contrary, GraphQL addresses highly relational or nested data by allowing clients to specify exactly which data they need in a single request. This makes it more flexible than REST in data-heavy applications. However, this flexibility causes challenges with caching, implementation complexity, and performance in certain use-cases. [25] Additionally, GraphQL performs well in applications with varied client requirements such as mobile, desktop, and web interfaces consuming the same API. However, it may not be ideal for systems requiring simple, predictable responses. [26]

Thirdly, gRPC focuses on performance and efficiency. It is particularly suited to system-to-system communication within microservices, where speed, reliability, and strict contracts are essential such as when an authentication service needs to communicate quickly and consistently with a payment service in a large distributed system. [27] However, its lack of native browser support limits its suitability for public web APIs. As well as this, its use of binary formats makes debugging more complicated. Lastly, gRPC also supports streaming, allowing continuous two-way data transfer between clients and servers, which is a capability not offered natively by REST.

To conclude, REST remains the most appropriate choice for simple web APIs, while GraphQL and gRPC serve more specialised roles.

Use-cases

REST is ideal for traditional web services that require simplicity, standardisation, and broad compatibility. Example use-cases include online shopping systems, blogs, booking services, and public data APIs where predictable endpoints and simple data retrieval are the main requirements. [28]

Use cases for GraphQL include social media platforms, dashboards, and mobile applications. This is because it is well suited to applications with complex or highly variable data access needs. These use cases often require data from multiple related sources and benefit from GraphQL's single endpoint querying and precise data selection. [29]

Furthermore, gRPC is most effective in microservices, real-time systems, and internal enterprise environments. High-performance use-cases such as video streaming pipelines, IoT networks, financial trading systems, and analytics platforms benefit from gRPC's speed, binary serialisation, and streaming capabilities because these features allow rapid, reliable, and continuous communication between services, even under heavy loads [30]

Conclusion

To conclude, REST, GraphQL, and gRPC each offer distinct approaches to building APIs. REST is simple and widely supported but can struggle with complex data retrieval, while GraphQL provides precise data querying at the cost of greater implementation complexity. gRPC excels in high-performance internal service communication but is less suitable for traditional web APIs. Thus, the choice of protocol depends on the application's requirements, environment, and data complexity.

Task 2

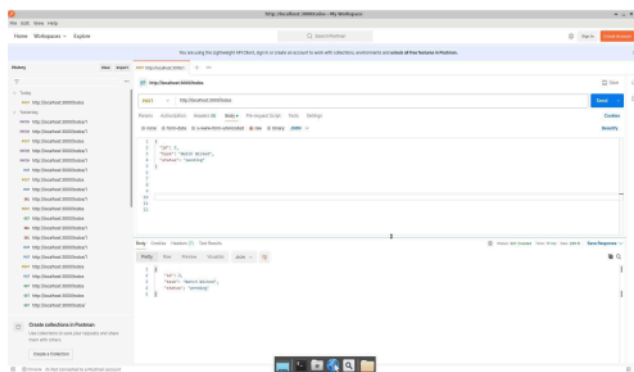
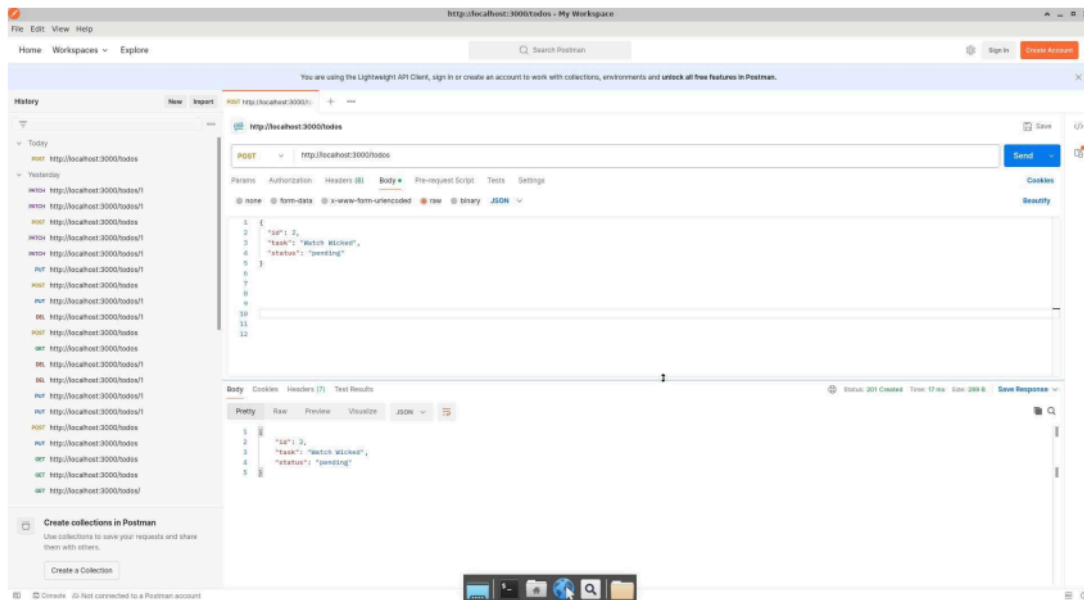
Introduction

This task outlines the workflow and tests for a to-do list API, demonstrating how a task progresses through its lifecycle using standard HTTP methods. The workflow highlights how CRUD functions support task creation, status changes, and overall system behaviour.

Workflow Concept for To-Do List

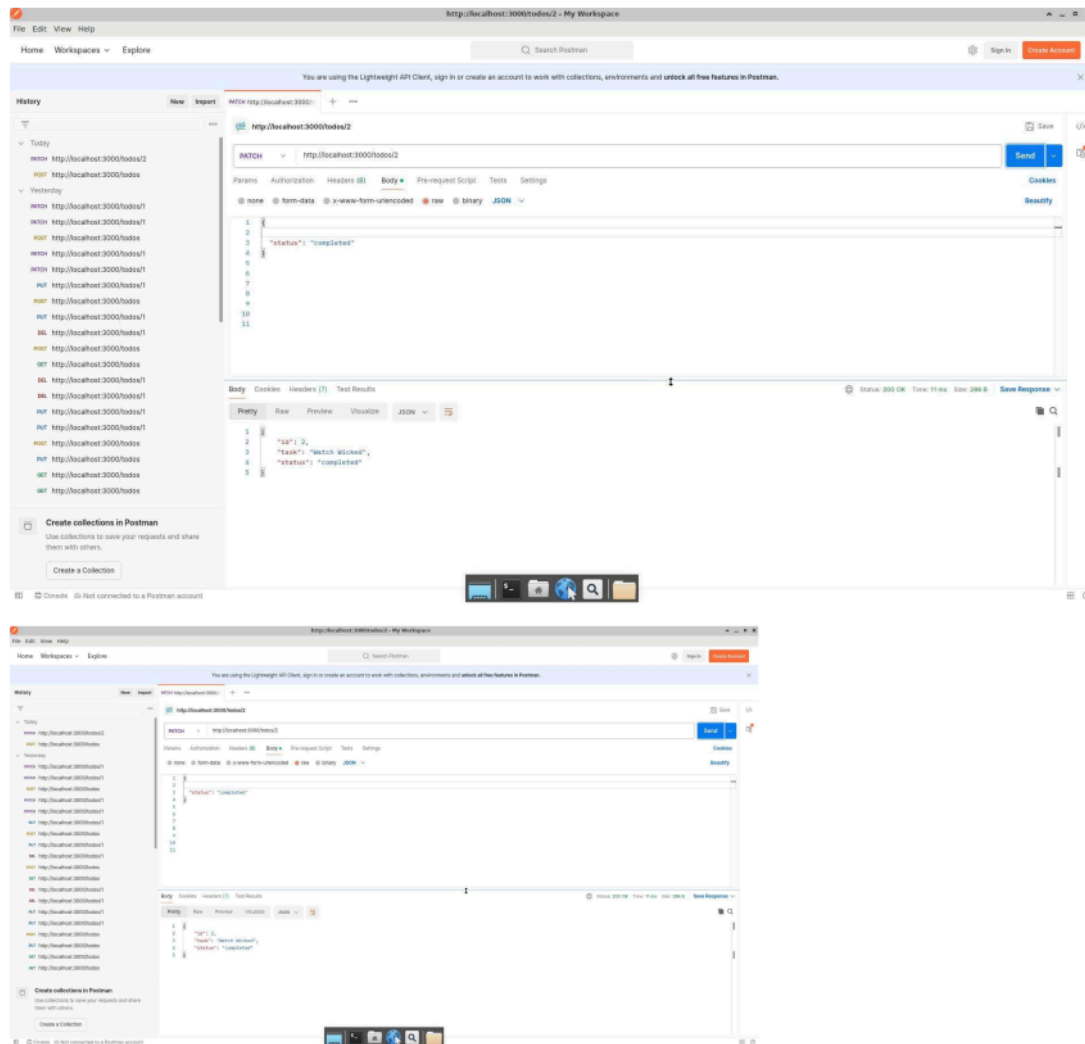
Stage 1 – Create task

When a user creates a new to do via POST, it enters the system with a status of "pending" as shown below.



Stage 2 — Task Completed

When the user completes the task, they PATCH the to do to change its status to "done" as shown below:



This workflow consists of two stages: first, when the to do is created, it enters the system with a status of 'pending'. Second, when the user completes the task, the status is updated to 'completed' using a PATCH request. This demonstrates a simple lifecycle workflow for tasks in the system.

Software Testing Evidence - To-Do List API

All tests were documented in a structured table, detailing test setup, steps, expected outcomes, and actual outcomes. This approach ensures that the API is robust, reliable,

and behaves as intended under various scenarios, providing clear evidence of effective software testing practices.

Test Title	Test Author	Test Description	Test Setup	Steps	Expected Outcome	Actual Outcome
Add a new to do	Seren a Jonathan	User adds a new to do task using POST to dos endpoint	Server running, Postman open	1. Open Postman. 2. Set method to POST. 3. URL: http://localhost:3000/todos . 4. Body → raw → JSON: { "task": "Buy groceries" }. 5. Click Send	To do is added to the system with status: "pending". JSON response includes the new id, task, and status.	To do is added successfully. JSON response: { "id": 1, "task": "Buy groceries", "status": "pending" }
View all to dos	Seren a Jonathan	User retrieves all to dos using GET /to dos	Server running, at least 1 to do exists	1. Set method to GET. 2. URL: http://localhost:3000/todos . 3. Click Send	Response returns an array of all to dos currently in the system.	Response returned [{ "id": 1, "task": "Buy groceries", "status": "pending" }]
Update a to do fully (PUT)	Seren a Jonathan	User updates a to do completely using PUT /todos/:id	Server running, to do with id: 1 exists	1. Set method to PUT. 2. URL: http://localhost:3000/todos/1 . 3. Body → raw → JSON: { "task": "Buy milk and bread", "status": "done" }. 4. Click Send	To do with new tasks and status. Response returns updated todo	JSON response: { "id": 1, "task": "Buy milk and bread", "status": "done" }
Update a to do partially	Seren a Jonathan	User updates a to do partially using	Server running, to do with	1. Set method to PATCH. 2. URL: http://localhost:3000/todos/1 . 3. Body → raw →	Only the provided field (status) is updated.	JSON response: { "id": 1, "task": "Buy milk

(PATCH)		PATCH /todos/:id	id: 1 exists	JSON: { "status": "done" }. 4. Click Send	Response returns updated to do	and bread", "status": "done" }
Delete a to do	Seren a Jonathan	User deletes a to do using DELETE /todos/:id	Server running, to do with id: 1 exists	1. Set method to DELETE. 2. URL: http://localhost:3000/todos/1 . 3. Click Send	To do is removed from the system. Returns a confirmation message	JSON response: { "message": "Todo 1 deleted." }
Workflow test: Pending → Done	Seren a Jonathan	Demonstrate the workflow of a to do moving from pending to done	Server running, to do with id: 2 exists	1. POST a new to do { "task": "Finish assignment" } → pending. 2. PATCH todo /todos/2 → { "status": "done" }	Todo initially created with pending status, then updated to done	JSON response: { "id": 2, "task": "Finish assignment", "status": "pending" } → { "id": 2, "task": "Finish assignment", "status": "done" }

Software Testing Process for To-Do List API

The API developed for this assignment implements a simple to-do list with full CRUD functionality (Create/POST, Read/GET, Update/PUT, Delete). Evidence of software testing was collected using Postman as shown prior. Following this was the table, offering a structured approach to testing to ensure that each endpoint functions correctly.

GET

One function tested was the GET operation. This operation retrieves all to-do items stored in the system. The endpoint was checked to ensure that it returns an array of tasks in the

correct JSON format, including any previously added items. The outcome of the test is shown below that confirmed that data types, keys, and values matched the expected structure.

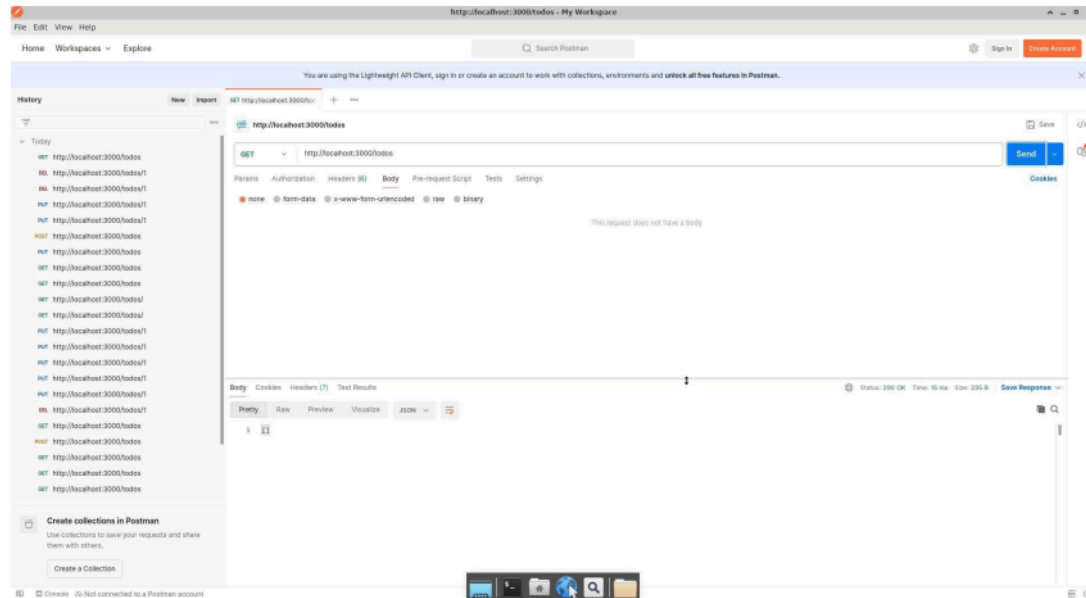


Figure 1: GET Test

Create/POST

Another function tested was the Create/POST when adding a new task. A JSON payload specifying the task description was submitted to the server. The test confirmed that a new to do item was created with a unique ID and a default status of "pending." The test outcome is shown below presenting the desired result.

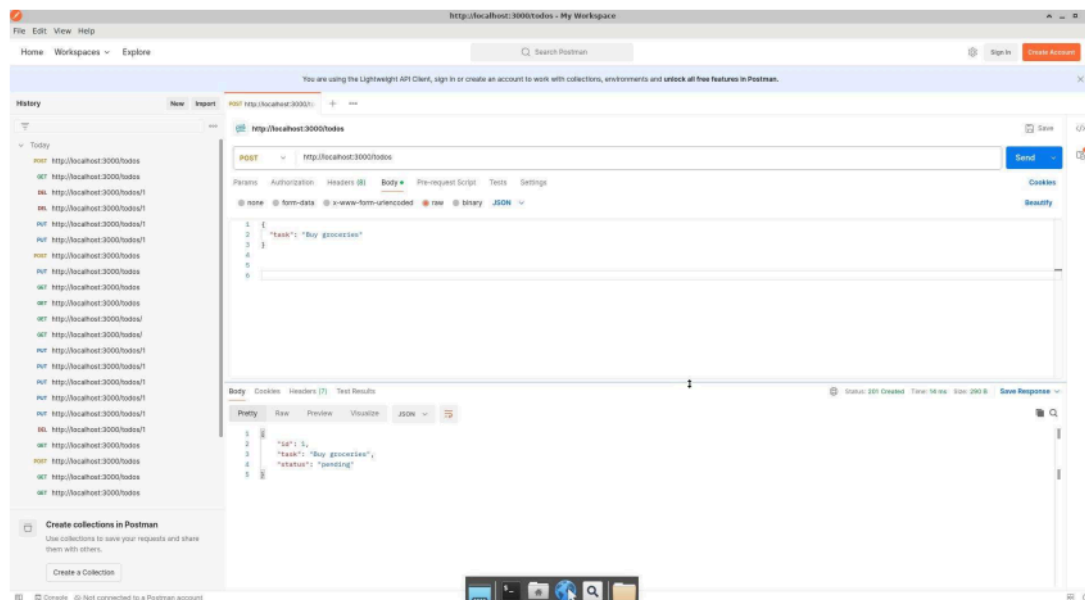


Figure 3: POST Test

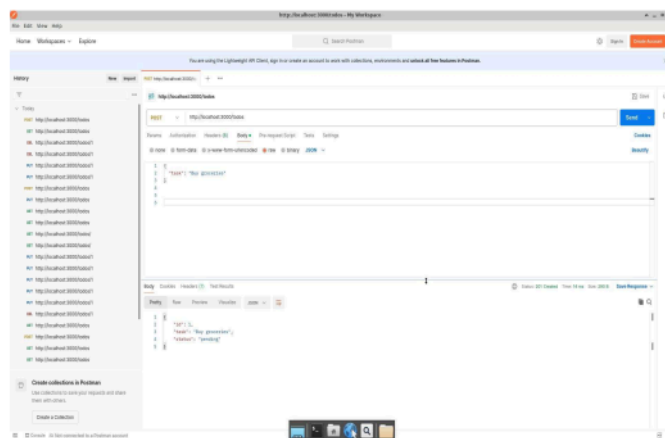


Figure 4: POST Test More Detail

PUT/Update

The PUT operation was also tested to allow complete replacement of an existing to-do. By sending a JSON payload with the updated task and status to the endpoint specifying the to-do ID, it was verified that the server correctly replaced the previous data. Any discrepancies in the response would indicate a flaw in the API logic. However, as shown in

the figure below, the desired outcome was received, which is the original to-do being fully updated with the new task and status, confirming the PUT operation functions correctly.

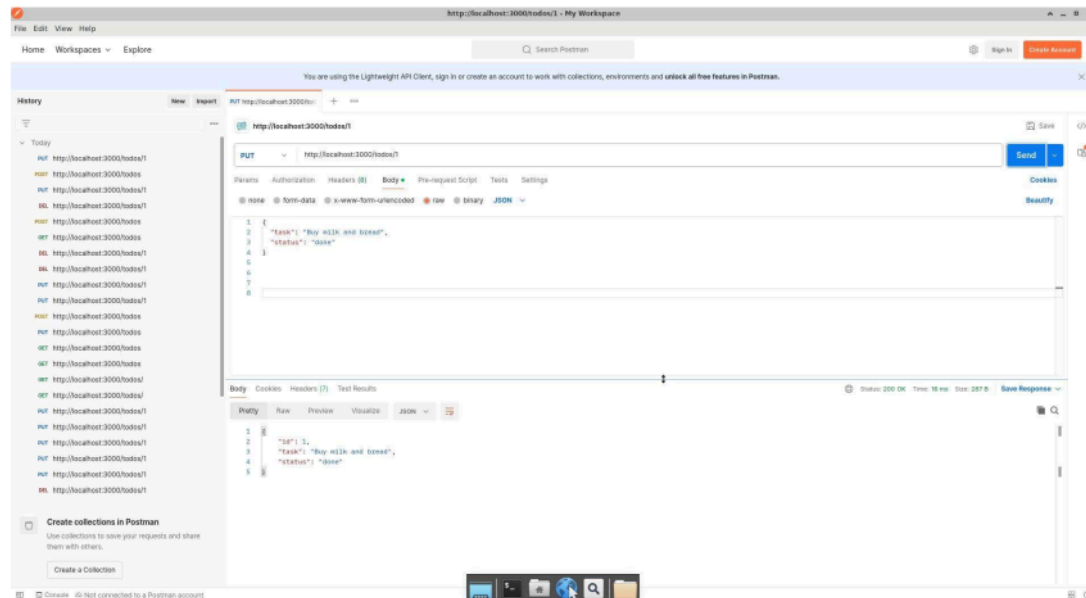


Figure 5: PUT Test

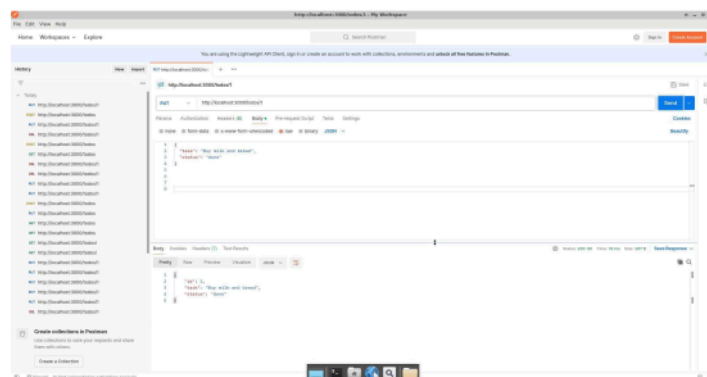


Figure 6: PUT Test More Detail

PATCH

Partial updates were tested using the PATCH operation, which allows modification of specific fields only which is demonstrated with the figures below. The endpoint

successfully updated the provided fields while leaving other data unchanged, demonstrating flexibility and preventing accidental overwriting of existing information.

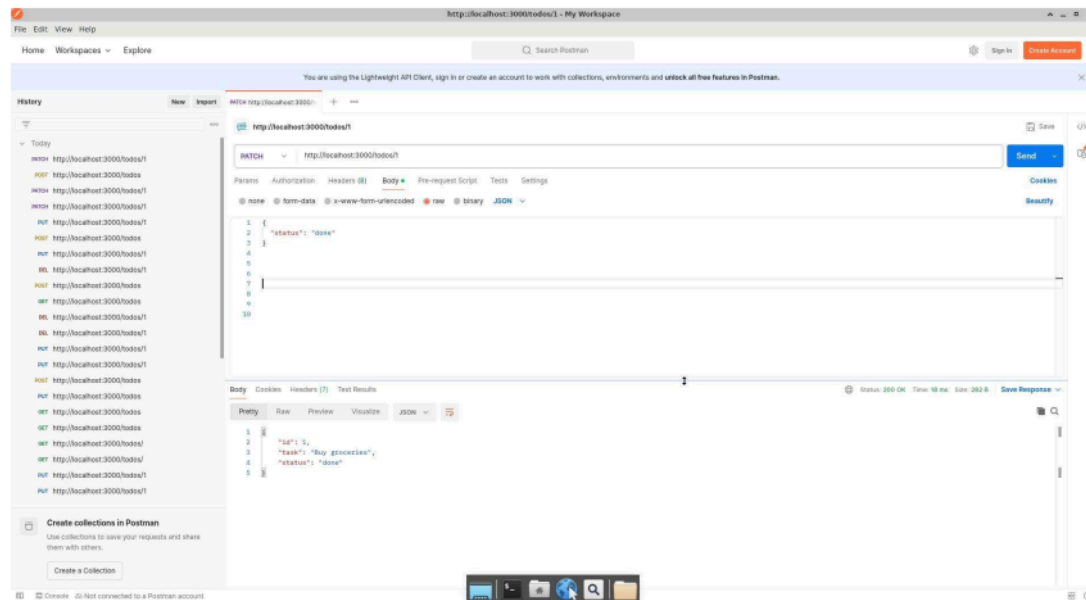


Figure 7: PATCH - “status” Test

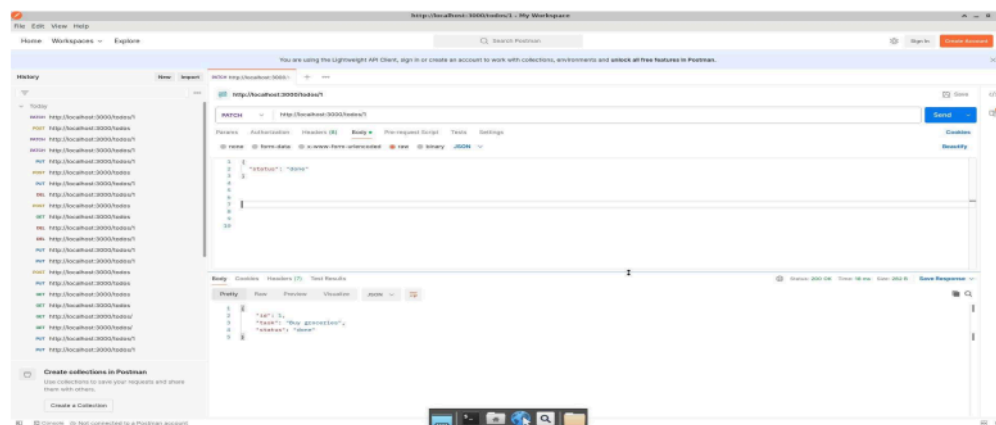


Figure 8: PATCH - “status” More Detail Test

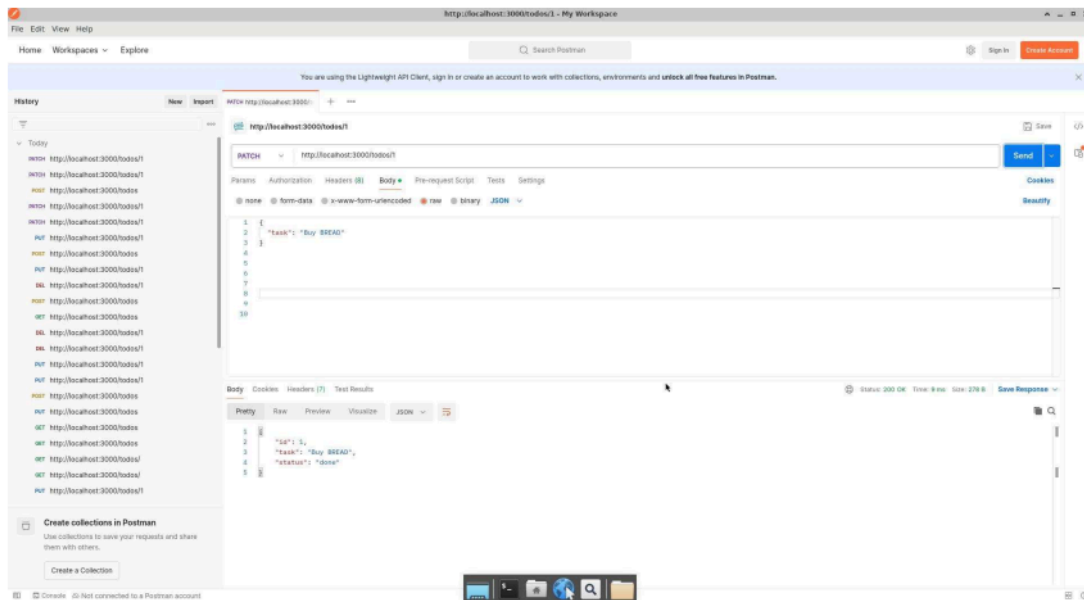


Figure 9: PATCH - "task" Test

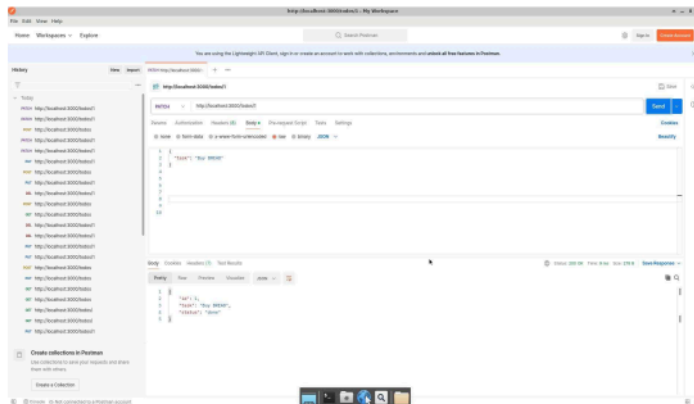


Figure 10: PATCH - "task" More Detail Test

DELETE

The DELETE operation was also tested. This function removed a specific to-do item based on its ID. The endpoint was tested for both existing and non-existent items. Successful

deletions returned confirmation messages, while attempts to delete non-existent items returned appropriate error messages, confirming correct handling of both scenarios.

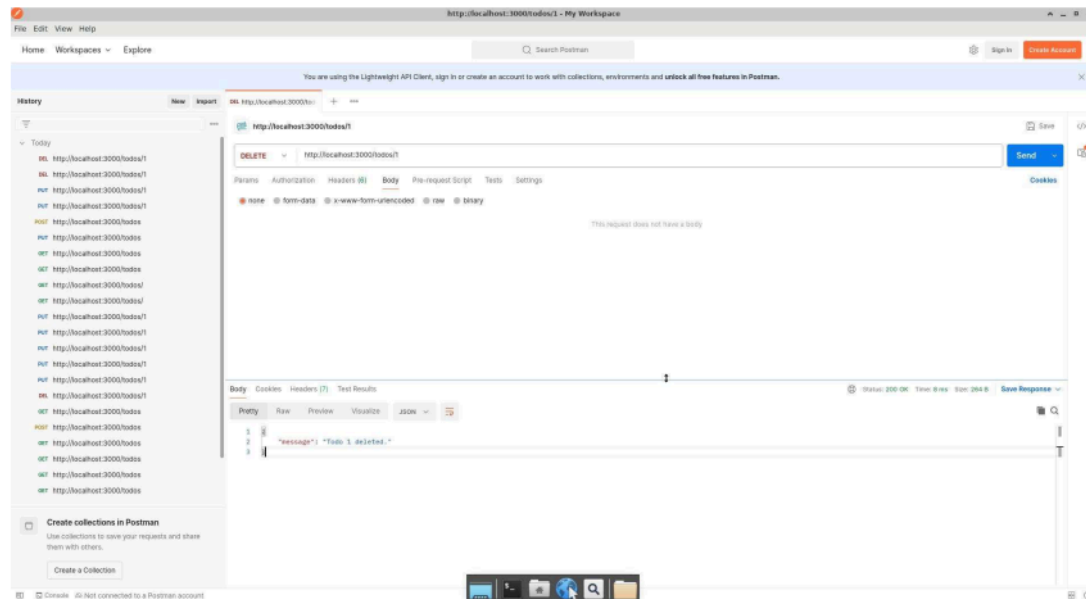


Figure 11: DELETE Test

Conclusion

Overall, the to-do list API successfully demonstrates essential CRUD operations, workflow processing, and structured software testing. The testing confirms that each endpoint performs as intended, providing clear evidence of reliable and consistent API behaviour within the system.

Bibliography

- [1] B. Cooksey, B. Landers, and D. Schreiber, "An Introduction to APIs," 2014. Available: https://cdn.zapier.com/storage/learn_ebooks/e06a35cfcf092ec6dd22670383d9fd12.pdf (accessed Nov. 18, 2025).
- [2] T. Gaitatzis, *Learn REST APIs*. Tony Gaitatzis, 2019. Accessed 18/11/2025.
- [3] O. Ali, "Popular API Technologies: REST, GraphQL, and gRPC," *Theseus.fi*, 2024, doi: <https://www.theseus.fi/handle/10024/864399>. (accessed Nov. 18, 2025).
- [4] S. Guha and S. Majumder, "", *SSRN Electronic Journal*, 2025, doi: <https://doi.org/10.2139/ssrn.5289281>. (accessed Nov. 18, 2025).
- [5] "Design and implementation of a REST API-based client-server architecture for multi-sensor IoT monitoring - ProQuest," *Proquest.com*, 2025.
<https://www.proquest.com/openview/0e4cf5abff7398098a5b789e44007504/1?pq-origsite=gscholar&cbl=2037694> (accessed Nov. 18, 2025).
- [6] "Web API Development with ASP.NET Core 8," *Google Books*, 2024.
<https://books.google.co.uk/books?hl=en&lr=&id=bhb5EAAAQBAJ&oi=fnd&pg=PP1&dq=REST+APIs+use+standard+HTTP+methods+to+manipulate+resources:+GET+retrieves+data> (accessed Nov. 18, 2025).
- [7] M. Coblenz, W. Guo, Kamatchi Voozhian, and J. S. Foster, "A Qualitative Study of REST API Design and Specification Practices," Oct. 2023, doi: <https://doi.org/10.1109/vl-hcc57772.2023.00025>. (accessed Nov. 19, 2025).
- [8] S. Guha and S. Majumder, "A Comparative Study between Graph-QL& Restful Services in API Management of Stateless Architectures," *International Journal on Web Service Computing*, vol. 11, no. 2, pp. 1–16, Jun. 2020, doi: <https://doi.org/10.5121/ijwsc.2020.11201>. (accessed Nov. 19, 2025).
- [9] N. Swanson, "The Evolution of API Architectures; REST & GraphQL," 2024. Accessed: Nov. 19, 2025. [Online]. Available: <https://scholarsbank.uoregon.edu/server/api/core/bitstreams/e21656fa-73f8-46c0-bfc6-cb573bdeee2b/content>
- [10] S. Kanthed, "Rest vs. GraphQL: Comparative Analysis of API Design Approaches," *International Journal of Multidisciplinary Research and Growth Evaluation.*, vol. 4, no. 1, pp. 984–991, 2023, doi: <https://doi.org/10.54660/ijmrge.2023.4.1.984-991>. Accessed: Nov. 19, 2025.

- [11] M. D. Scese, "DESIGN AND IMPLEMENTATION OF A GRAPHQL MESH GATEWAY: FEDERATING API ENDPOINTS BASED ON A DEFINED DATA MODEL," Jan. 2023, doi: <https://doi.org/10.37099/mtu.dc.etr/1651>. Accessed: Nov. 20, 2025.
- [12] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, "GraphQL: A Systematic Mapping Study," *ACM Computing Surveys*, Sep. 2022, doi: <https://doi.org/10.1145/3561818>. Accessed: Nov. 20, 2025.
- [13] Nagaraju Thallapally, "Enhancing Data Query Flexibility with GraphQL: Implementation and Best Practices," *Journal of Computer Science and Technology Studies*, vol. 6, no. 2, pp. 176–182, Jun. 2024, doi: <https://doi.org/10.32996/jcsts.2024.6.2.20>. Accessed: Nov. 20, 2025.
- [14] P. Mandani, L. R. B K, N. R. Shetty, and R. TN, "A Comprehensive Analysis of GraphQL," *SSRN Electronic Journal*, 2024, doi: <https://doi.org/10.2139/ssrn.4913802>. Accessed: Nov. 21, 2025.
- [15] "The Eurasia Proceedings of Science, Technology, Engineering and Mathematics," *Epstem.net*, 2025. <https://www.epstem.net/index.php/epstem> (accessed Nov. 21, 2025).
- [16] A. John, "Exploring GraphQL as an Alternative to REST," *www.theseus.fi*, 2023. <https://www.theseus.fi/handle/10024/816907> (accessed Nov. 22, 2025).
- [17] O. Ali, "Popular API Technologies: REST, GraphQL, and gRPC," *Theseus.fi*, 2024, doi: <https://www.theseus.fi/handle/10024/864399>. (accessed Nov. 22, 2025).
- [18] E. Obuse, E. D. Erigha, B. P. Okare, A. C. Uzoka, S. Owoade, and N. Ayanbode, "Optimizing Microservice Communication with gRPC and Protocol Buffers in Distributed Low-Latency API-Driven Applications," *International Journal of Multidisciplinary Futuristic Development*, vol. 1, no. 1, pp. 45–55, 2020, doi: <https://doi.org/10.54660/ijmfd.2020.1.1.45-55>. (accessed Nov. 22, 2025).
- [19] V. Muniyandi, "Utilizing Grpc For High-Performance Inter-Service Communication In .NET," *SSRN Electronic Journal*, 2025, doi: <https://doi.org/10.2139/ssrn.5381441>. (accessed Nov. 22, 2025).
- [20] L. Geng *et al.*, "RR-Compound: RDMA-Fused gRPC for Low Latency, High Throughput, and Easy Interface," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 8, pp. 1488–1505, May 2024, doi: <https://doi.org/10.1109/tpds.2024.3404394>. (accessed Nov. 22, 2025).
- [21] N. Hamo and S. Saberian, "Evaluating the performance and usability of HTTP vs gRPC in communication between microservices," *DIVA*, 2023. <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1768795&dsid=9842> (accessed Nov. 23, 2025).

- [22] JamesNK, "Compare gRPC services with HTTP APIs," *Microsoft.com*, Jul. 31, 2024. <https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-10.0> (accessed Nov. 23, 2025).
- [23] N. Hedelin, "Benchmarking and performance analysis of communication protocols : A comparative case study of gRPC, REST, and SOAP," *DIVA*, 2024. <https://www.diva-portal.org/smash/record.jsf?dswid=9842&pid=diva2%3A1887929> (accessed Nov. 23, 2025).
- [24] S. Chandra and A. Farisi, "Comparative Analysis of RESTful, GraphQL, and gRPC APIs: Performance Insight from Load and Stress Testing," *Jurnal Sisfokom (Sistem Informasi dan Komputer)*, vol. 14, no. 1, pp. 81–85, Jan. 2025, doi: <https://doi.org/10.32736/sisfokom.v14i1.2315>. (accessed Nov. 23, 2025).
- [25] Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin, and I. Aswad, "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC," *International Journal of Electronics and Telecommunication*, vol. 70, no. 2, pp. 429–436, 2024, Available: <https://ijet.ise.pw.edu.pl/index.php/ijet/article/view/10.24425-ijet.2024.149562> (accessed Nov. 24, 2025).
- [26] "The Eurasia Proceedings of Science, Technology, Engineering and Mathematics," *Epstem.net*, 2025. <https://www.epstem.net/index.php/epstem> (accessed Nov. 24, 2025).
- [27] Ritu, S. Arora, A. Bhardwaj, A. Kukkar, and S. Kaur, "A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice- Based Ecosystems," 2024 *International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*, pp. 621–626, May 2024, doi: <https://doi.org/10.1109/innocomp63224.2024.00107>. (accessed Nov. 24, 2025).
- [28] A. Singh, "What is Rest API?: Understanding REST Architecture with Examples | BrowserStack," *BrowserStack*, Dec. 20, 2024. <https://www.browserstack.com/guide/rest-api> (accessed Nov. 25, 2025).
- [29] Kong, "What is GraphQL? Use Cases and Examples," *Kong Inc.*, 2023. <https://konghq.com/en-gb/blog/learning-center/graphql> (accessed Nov. 25, 2025).
- [30] R. Rajpal, "What is gRPC: Features, Architecture, and Use Cases | BrowserStack," *BrowserStack*, Oct. 22, 2025. <https://www.browserstack.com/guide/what-is-grpc> (accessed Nov. 26, 2025).

Assignment 1 - Cloud Based Web Services - Serena Jonathan - 2306363

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

74/100

Generic Assessment Criteria	Marks available	Marks awarded
1. Engagement with Literature Skills .	Assignment 1: 5	
Good number of sources, reasonable range of credible and academic sources as well as websites. References correctly used through the text.		3
2. Knowledge and Understanding Skills	Assignment 1: 10	
Investigation, use-cases and advantages / disadvantages. Good knowledge demonstrated of REST, GraphQL and gRPC. A little more detailed		9

discussion in the workings would have been beneficial. A nicely detailed discussion of comparisons and use-cases. A nice narrative through the technology, comparisons and use cases.		
1. Cognitive and Intellectual Skills	Assignment 1: 30	
Design and documentation Test documentation and testing Good use of test table, although a further box for PASS or FAIL would have been a desirable addition. Some reasonable discussion of the api and the endpoints and nice use of both PUT and PATCH.		2 4
4. Practical Application Skills	Assignment 1: 50	
AJAX implementation REST API A nice example of a mostly functional REST API. Commented and reasonably packaged. A well packaged folder with package.json. A readme file may have been advantageous and would have cut		3 4. 5

down the word count in the report.		
Possibly some way to		
5. Transferable Skills for Life and Professional Practice	Assign ment 1: 5	
Presentation		3. 5
Reasonable presentation, clear introduction and conclusion.		
All images / figures should have been captioned.		

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21
