

Final

by Luke Redmore

Submission date: 03-Dec-2025 06:40PM (UTC+0000)

Submission ID: 269488584

File name: 41367_Luke_Redmore_Final_708807_388459959.docx (550K)

Word count: 2759

Character count: 16803



CLOUD WEB SERVICES

CW-1

Investigation into advantages of a RESTful web
API and implementation with consideration to
alternatives

Abstract

An investigation into the usefulness of REST as both an architecture and as a design standard followed along by a brief dive into SOAP and GraphQL, as well as an implementation of a RESTful webservice.

Luke Redmore

2219318@student.uwtsd.ac.uk

Contents

REST, RESTful and its alternatives	2
REST	2
REST briefly summarized	2
RESTful API fundamentals and advantages	2
Collated examples for RESTful URIs:	2
Key advantages of REST	3
Potential drawbacks	3
Workaround for RESTful applications to handle real time updates	3
Alternatives to REST	3
How they stack up, and use cases of each	4
Implementation of a RESTful API:	5
Overview	5
Express server setup with essential dependencies:	6
Endpoint(s)	6
POSTman endpoints to test and use:	8
References	15

REST, RESTful and its alternatives

REST

The last thirty or so years has seen many paradigms rise and fall, particularly around application programming interfaces (APIs) as businesses identify the crucial role that APIs play in customer experience and seamless service delivery, they can be developed to be more interoperable. One such paradigm is REST or RESTful APIs which naturally stands for Representational State Transfer.

REST briefly summarized

Representational State Transfer (REST) Is best explained as a software architecture that imposes a set of conditions on how an API should be implemented, REST's origins can be traced back as a guideline to manage communication across a complicated network such as the internet. [1]

RESTful API fundamentals and advantages

RESTful APIs, as defined by Roy Fielding in 1999, follow a set of principles that simplify implementation and scalability.

Data is represented by its format (e.g., JSON, XML) rather than physical files, and all data is treated as resources. Each resource should have a unique identifier accessible via a URI. While not mandatory, URIs are often human-readable and self-descriptive, even though they are primarily accessed by software. Resource access and manipulation should use standard HTTP methods such as **GET**, **POST**, and **PUT** which can be expressed as a uniform interface. [2]

Collated examples for RESTful URIs:

<https://books.google.co.uk/books?>

[/developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching](https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching)

REST is by design, stateless. As such each HTTP request should be independent, meaning any resource modification happens in isolation and completes fully or not at all—avoiding partial updates this is often referred to as an atomic operation.

In production environments, requests often pass through load balancers, so storing session state on the server is impractical because clients may not hit the same server twice. If state must be preserved, it should be encoded in a RESTful way, such as within the URI.

This stateless design enables scalability and high availability. Clients remain unaffected by infrastructure changes, as multiple servers can seamlessly handle requests even when nodes are added or removed. [2]

Key advantages of REST

The use of standard HTTP methods facilitates easy implementation and understanding in lieu of additional protocols, further levying the HTTP stack REST can take full advantage of HTTP caching which can store and reuse a stored request to prevent the client from having to be in direct contact with the origin server for each request which greatly reduces the response time for frequent requests.

Due to REST requests being atomic in nature they are far more reliable as partial updates are not possible aiding data integrity even in distributed environments. REST architecture also boasts incredible availability and immense scalability, this is primarily due to the clients not relying on the server side allowing numerous servers to seamlessly serve requests intermittently following on from this statelessness allows the servers to handle requests independently, which further enables horizontal scaling through load balancers and distributed systems.

Potential drawbacks

REST by design requires that every request must include all the necessary information as the server does not store client state, this could include authentication details such as the users' roles as well as the users' details. This will have to be done for each, and every subsequent request made for a resource which can lead to "statelessness overhead" as the payloads will be far larger and require extra processing for extra requests.

Furthermore, REST poses a limited use case for Real-Time capabilities without workarounds, for example its un-ideal for real-time updates such as chat applications or live streaming as REST is request-response only meaning the client must initiate communication [3]

Workaround for RESTful applications to handle real time updates

As is often the case for mainstream paradigms workarounds are created to adapt to use cases that weren't otherwise possible.

One such case for REST is the implementation of web-sockets, these being a protocol designed to facilitate full-duplex communication allowing bi-directional data transfer, while maintaining a persistent connection as once a WebSocket is bound and opened it remains so. This removes the need for the client to continuously open a new HTTP connection with the server greatly reducing HTTP request overhead and by consequence latency effectively undoing the primary pitfall of a request-based architecture. [15][16]

Alternatives to REST

Although REST (Representational State Transfer) is widely adopted for building modern web services, it is not the only mechanism for exchanging data between distributed systems. Two important alternatives are Simple Object Access Protocol (SOAP) and GraphQL, each offering unique capabilities that suit specific technical requirements.

SOAP is a protocol-driven approach to building web services. Unlike REST, which is an architectural style, SOAP defines strict standards for message formatting, typically using XML

exclusively. SOAP services often include a WSDL (Web Services Description Language) file that acts as a formal contract describing operations and expected message structures [4].

SOAP's rigid and standardised structure provide several advantages, particularly in environments where reliability and security are critical. Because SOAP supports the WS- such as WS-Security, WS-Addressing, and WS-Reliable Messaging. it enables features like encrypted communication, digital signatures, and guaranteed message delivery. These capabilities make SOAP well suited to large organisations that follow strict industry regulations, where transactional integrity and formal contracts are a priority. Industries such as banking, government, and telecommunications often rely on SOAP because its protocol ensures consistent, validated communication between distributed systems [5].

However, SOAP's strictness also introduces limitations. XML messages are typically more verbose than JSON payloads used by REST, meaning SOAP requires greater bandwidth and can be slower to process. The need to work with WSDL files, complex schemas, and extensive security configurations can increase development and maintenance overhead. Additionally, SOAP services do not naturally benefit from the lightweight, cache-friendly behaviour offered by RESTful HTTP operations, making them less practical for modern web and mobile applications where performance and low latency are essential [6].

In contrast to SOAP's formality, GraphQL represents a more flexible and modern approach to data exchange. Developed by Facebook, GraphQL is a query-language-based API model that allows clients to request exactly the data they need [7]. Unlike REST, which generally requires multiple endpoints for different resources, GraphQL typically exposes a single endpoint through which clients define the structure of the response using highly customisable queries. This reduces issues such as over-fetching and under-fetching, which are common when REST endpoints return fixed data structures [8].

GraphQL's flexibility is particularly beneficial in applications with dynamic interfaces or where bandwidth is limited, such as mobile platforms. By allowing multiple related pieces of information to be retrieved in a single request, GraphQL reduces the number of roundtrips needed between the client and server. Its strongly typed schema also enhances developer tooling, enabling features such as auto-completion, schema introspection, and interactive query explorers. These characteristics make GraphQL popular in modern frontend systems and data-intensive environments where efficient, predictable data retrieval is crucial [9].

Despite its strengths, GraphQL also presents challenges. A significant drawback is the added server-side complexity required to resolve nested queries and ensure that expensive requests do not overwhelm system resources. GraphQL's single-endpoint structure also makes traditional HTTP caching techniques less effective, requiring developers to implement custom caching strategies or rely on specialised client libraries. Additionally, without strict query constraints, poorly designed GraphQL queries can place heavy load on backend systems, making careful schema design and performance management essential [10].

How they stack up, and use cases of each

REST is commonly used in general-purpose web and mobile applications due to its simplicity, scalability, and natural fit with HTTP. It is well suited for CRUD-based systems such as social

media platforms, e-commerce websites, and public APIs. Its lightweight JSON responses and compatibility with caching make REST ideal for high-traffic systems that must handle millions of requests efficiently.

SOAP is typically chosen for systems that require guaranteed message delivery, strict security controls or formalised communication contracts. Sectors such as banking, insurance, healthcare, and government often use SOAP because WS standards support encrypted communication, digital signatures, and transaction reliability [12]. It is especially valuable in mission-critical systems such as payment processing or identity verification networks.

GraphQL is most effective in applications with complex, highly relational data requirements or rapidly changing user interfaces. It is widely used in modern single-page applications, mobile platforms with tight bandwidth constraints, and systems that aggregate data from multiple microservices. Companies such as Facebook, Shopify, and Netflix use GraphQL to enable efficient, flexible data retrieval that adapts to diverse client needs. [13]

When comparing these alternatives to REST, the differences become clear. REST remains successful largely due to its simplicity, statelessness, and natural alignment with the HTTP protocol. Its lightweight JSON messages, ease of caching, and straightforward implementation make it a strong choice for most web and mobile applications. SOAP, while more complex, excels in systems where strong security, predictable contracts, and transactional guarantees are required. Meanwhile, GraphQL offers a high degree of flexibility and efficiency in data-driven applications but demands more careful server-side management. A performance comparison study between REST and GraphQL found that while REST provides faster response times possibly up to fifty percent faster while maintaining higher throughput (check source), GraphQL demonstrated significantly better resource efficiency, reducing CPU load and memory usage by over thirty seven percent [14].

Ultimately, the selection of REST, SOAP, or GraphQL depends on the specific requirements of the system being developed. Organisations prioritising security and structured communication may choose SOAP, those needing rapid and flexible data queries may opt for GraphQL, and those looking for a balanced, scalable approach often rely on REST especially considering effective workarounds such as WebSocket's.

Implementation of a RESTful API:

Overview

This project is designed to use N2YO's API to fetch satellite data via a sat Id and use data fields such as eclipsing and altitude relative to a hardcoded latitude and longitude currently set to UWTSD's IQ building. Currently this project handles five of the most tracked satellites and can

handle user observations via a simple file system as this enables an in-memory approach of storing data as opposed to coupling a more complex database system.

Express server setup with essential dependencies:

```
const express = require("express");
const cors = require("cors");
const fetch = require("node-fetch");
// file system makes creation and writing to my in memory file easier
const fs = require("fs");

require("dotenv").config();

const app = express();
const PORT = process.env.PORT;

app.use(cors());
app.use(express.json());
```

The code seen above sets up an express server and its associated middleware, the 'require("stuff")' statements can be best thought of as dependencies and are installed in the package.json

This project sets up api keys and a base url for N2YO as well as port and app as environmental constants. This is brought into scope via;

```
require("dotenv").config();    // Loads .env file variables
const PORT = process.env.PORT; // Gets port from environment
```

the middleware stack:

```
app.use(cors());    // Enables cross-origin requests
app.use(express.json()); // Parses JSON request bodies
```

'cors' will be used to allow the frontend to eventually call the api without the browser blocking it.

'express.json' automatically parses JSON from request bodies allowing req.body to be used

'node-fetch' facilitates the fetching of data via HTTP request from N2YO.

Endpoint(s)

```

// GET - Get all satellites with real-time positions
app.get("/api/satellites", async (req, res) => {
  try {
    // Get user location from query parameters
    const { lat, lng, location } = req.query;
    const userLat = lat ? parseFloat(lat) : 51.6214;
    const userLng = lng ? parseFloat(lng) : -3.9436;

    const satellitesWithPositions = [];
    // Iterate through all satellites via name and id from SATELLITES object
    for (const sat of SATELLITES) {
      console.log(
        `Fetching data for ${sat.name} (ID: ${sat.id}) at ${userLat},${userLng}...`,
      );
      const position = await getSatellitePosition(sat.id, userLat, userLng);
      // Create a standard object to store incoming satellite data with
      // Optional chaining '?' to safe access lat and long and provide defaults if not set
      const satelliteData = {
        id: sat.id,
        name: sat.name,
        latitude: position?.latitude || 0,
        longitude: position?.longitude || 0,
        altitude: position?.altitude || 0,
        visible: position?.visible || false,
        // If position is not null Api call succesful else failed
        dataAvailable: position !== null,
      };
      // Push the standard object onto satellitesWithPositions array
      satellitesWithPositions.push(satelliteData);
      console.log(`${sat.name}: ${position ? colors.green + "Success" + colors.reset : colors.red + "Failed" + colors.reset}`);
    }

    res.json(satellitesWithPositions);
  } catch (error) {
    console.error(colors.red + "Error fetching satellites:", error + colors.reset);
    res.status(500).json({ error: "Failed to fetch satellite data" });
  }
});

```

```

// GET - Get all observations
app.get("/api/observations", (req, res) => {
  // access helper function
  const data = readData();
  res.json(data.observations);
});

```

```

// POST - Create new observation plan
app.post("/api/observations", (req, res) => {
  const { satelliteId, satelliteName, plannedTime, userLocation } = req.body;

  if (!satelliteId || !satelliteName || !plannedTime) {
    return res.status(400).json({ error: "Missing required fields" });
  }
});

```

```
// DELETE - Delete observation
app.delete("/api/observations/:id", (req, res) => {
  const { id } = req.params;
  const data = readData();

  const index = data.observations.findIndex((obs) => obs.id == id);
  // A negative one index would put it 'out of bounds' or simply NOT THERE
  if (index === -1) {
    return res.status(404).json({ error: "Observation not found" });
  }

  data.observations.splice(index, 1);
  writeData(data);

  res.json({ message: "Observation deleted successfully" });
});
```

```
// Enjoy the purty ASCII art
app.post("/api/seeAscii", (req, res) => {
  try {
    // Get ASCII art as array
    const asciiArt = getAsciiArtArray();

    console.log(colors.cyan + "Sending awesome ASCII art to endpoint..." + colors.reset);

    // Create response data with ASCII art in payload
    const responseData = {
      message: "ASCII art displayed!",
      timestamp: new Date().toISOString(),
      endpoint: "/api/seeAscii",
      asciiArt: asciiArt,
    };

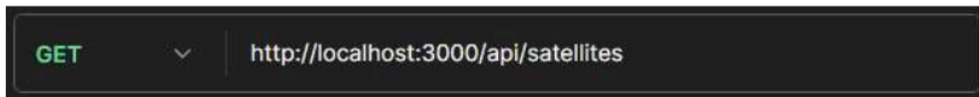
    // Send success response
    res.status(200).json(responseData);
  } catch (error) {
    console.error("Error displaying ASCII art:", error);
    res.status(500).json({ error: "Failed to display ASCII art" });
  }
});
```

POSTman endpoints to test and use:

Prior to attempting to access the applications endpoints, unpack and set the project up ensuring the .env file is in place as otherwise the data source will not be available.

Upon doing this please run;

'npm start' via the terminal.



Expected output;

```
[
  {
    "id": 25544,
    "name": "ISS (International Space Station)",
    "latitude": 0,
    "longitude": 0,
    "altitude": 0,
    "visible": false,
    "dataAvailable": false
  },
  {
    "id": 20580,
    "name": "Hubble Space Telescope",
    "latitude": 0,
    "longitude": 0,
    "altitude": 0,
    "visible": false,
    "dataAvailable": false
  },
  {
    "id": 43013,
    "name": "Starlink-1007",
    "latitude": 0,
    "longitude": 0,
    "altitude": 0,
    "visible": false,
    "dataAvailable": false
  },
  {
    "id": 43070,
    "name": "Starlink-1130",
```

```
"latitude": 0,  
"longitude": 0,  
"altitude": 0,  
"visible": false,  
"dataAvailable": false  
},  
{  
  "id": 28654,  
  "name": "Sentinel-1A",  
  "latitude": 0,  
  "longitude": 0,  
  "altitude": 0,  
  "visible": false,  
  "dataAvailable": false  
}  
]
```

(Run this prior to getting observations as the project has no observations pre-loaded)

POST	▼	http://localhost:3000/api/observations	Send	▼
------	---	--	------	---

{

GET	▼	http://localhost:3000/api/observations	Send	▼
-----	---	--	------	---

```
[  
{  
  "id": 1764697287974,  
  "satelliteld": 25544,  
  "satelliteName": "ISS",  
  "plannedTime": "2024-01-20T20:30:00",  
  "userLocation": "London, UK",  
  "status": "planned",  
  "createdAt": "2025-12-02T17:41:27.974Z"  
}
```

```
}  
]
```

PATCH



<http://localhost:3000/api/observations/1764697287974>

(This is also dependent on having an observation to update, note the id or numbers need to be updated to what is present in the observation i.e. "id": 1764697287974,)

```
{  
  "id": 1764697287974,  
  "satelliteld": 25544,  
  "satelliteName": "ISS",  
  "plannedTime": "2024-01-20T20:30:00",  
  "userLocation": "London, UK",  
  "status": "observed",  
  "createdAt": "2025-12-02T17:41:27.974Z",  
  "notes": "Clear sky, great view!", <- Gets added to original observation  
  "updatedAt": "2025-12-02T17:46:27.554Z"  
}
```

GET http://localhost:3000/api/observations Send

Docs Params Auth Headers (6) Body Scripts Settings Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body 200 OK · 6 ms · 528 B

{ } JSON Preview Visualize

```
1  [  
2    {  
3      "id": 1764697287974,  
4      "satelliteId": 25544,  
5      "satelliteName": "ISS",  
6      "plannedTime": "2024-01-20T20:30:00",  
7      "userLocation": "London, UK",  
8      "status": "observed",  
9      "createdAt": "2025-12-02T17:41:27.974Z",  
10     "notes": "Clear sky, great view!",  
11     "updatedAt": "2025-12-02T17:46:27.554Z"  
12   }  
13 ]
```

(You can cross reference this via the get endpoint for observations)

DELETE http://localhost:3000/api/observations/1764697287974

(The id parameter here also needs to be updated to what is present in the created observation)

Expected output:

```
{  
  "message": "Observation deleted successfully"
```

[illegible]

References

- [1] "View of Enhancing Data Query Flexibility with GraphQL: Implementation and Best Practices," *Al-kindipublisher.com*, 2025. <https://al-kindipublisher.com/index.php/jcsts/article/view/8830/7510> (accessed Nov. 28, 2025).
- [2] "RESTful Web API Design with Node.js," *Google Books*, 2016. https://books.google.co.uk/books?hl=en&lr=&id=cdJtDQAAQBAJ&oi=fnd&pg=PP1&dq=fundamentals+of+a+rest+api&ots=O-rHF1xTfU&sig=1LXlodGL_ct4cQwnmHQhe_OiupQ&redir_esc=y#v=onepage&q&f=true
- [3] "HTTP caching - HTTP | MDN," *MDN Web Docs*, Mar. 13, 2025. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching>
- [4] A. Soni and V. Ranga, "API Features Individualizing of Web Services: REST and SOAP," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 9S, pp. 664–671, Aug. 2019, doi: <https://doi.org/10.35940/ijitee.i1107.0789s19>.
- [5] S. Reddy, "Strategies for Integrating Services with External Systems Via Rest & Soap," *Universal library of engineering technology*, pp. 35–42, Mar. 2022, doi: <https://doi.org/10.70315/uloap.ulete.2022.005>
- [6] Kwong Yuen Chiu, M. Govindaraju, and R. Bramley, "Investigating the limits of SOAP performance for scientific computing," *High Performance Distributed Computing*, Jul. 2002, doi: <https://doi.org/10.1109/hpdc.2002.1029924>.
- [7] "GraphQL in Action," *Google Books*, 2021. https://books.google.co.uk/books?hl=en&lr=&id=1jszEAAAQBAJ&oi=fnd&pg=PR15&dq=graphql+api&ots=fyrAYXTDRE&sig=BK3EzIzpb6C2EsJhJMXtAFoaJlk&redir_esc=y#v=onepage&q&f=false (accessed Nov. 28, 2025).
- [8] N. Swanson, "The Evolution of API Architectures; REST & GraphQL," 2024. Accessed: Nov. 18, 2025. [Online]. Available: <https://scholarsbank.uoregon.edu/server/api/core/bitstreams/e21656fa-73f8-46c0-bfc6-cb573bdeee2b/content>
- [9] 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>.
- [10] S. Amareen, D. O. Soto, A. Dado, and A. Bosu, "GraphQL Adoption and Challenges: Community-Driven Insights from StackOverflow Discussions," *arXiv.org*, 2024. <https://arxiv.org/abs/2408.08363>
- [11] P. Gowda and A. N. Gowda, "Best Practices in REST API Design for Enhanced Scalability and Security," *Journal of Artificial Intelligence, Machine Learning and Data Science*, vol. 2, no. 1, pp. 827–830, Feb. 2024, doi: <https://doi.org/10.51219/jaimld/priyanka-gowda/202>.

[12] Sireesha Addanki, "Architectural Shifts in API Design: The Progression from SOAP to REST and GraphQL," *International Journal on Science and Technology*, vol. 16, no. 2, Jun. 2025, doi: <https://doi.org/10.71097/ijst.v16.i2.6579>

[13] "View of Enhancing Data Query Flexibility with GraphQL: Implementation and Best Practices," *Al-kindipublisher.com*, 2025. <https://al-kindipublisher.com/index.php/jcsts/article/view/8830/7510>

[14] A. Lawi, B. L. E. Panggabean, and T. Yoshida, "Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System," *Computers*, vol. 10, no. 11, p. 138, Oct. 2021, doi: <https://doi.org/10.3390/computers10110138>.

[15] I. Fette and A. Melnikov, *The WebSocket Protocol*, RFC 6455, Internet Engineering Task Force (IETF), Dec. 2011. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc6455.html>

[16] J. Fette and A. Melnikov, "Communicating and Displaying Real-Time Data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, Jul.–Aug. 2012, doi: 10.1109/MIC.2012.74.

FINAL GRADE

GENERAL COMMENTS

88/100

Generic Assessment Criteria	Marks available	Marks awarded
1. Engagement with Literature Skills	Assignment 1: 5	
Good range and number of sources, mostly credible and academic. References correctly used through the text.		5
2. Knowledge and Understanding Skills	Assignment 1: 10	
Investigation, use-cases and advantages / disadvantages. Some good understanding from the sources demonstrated of REST (and Web Sockets), SOAP and GraphQL.		9

Some good discussion of comparisons and how best used. Good detailed discussion of use-cases.		
3. Cognitive and Intellectual Skills	Assignment 1: 30	
Design and documentation Test documentation and testing Good discussion of the implementation of the API would have been desirable. Good documentation within the readme file demonstrates good cognitive understanding. Whilst evident in discussion, evidence of testing should have been shown within the report.		25
4. Practical Application Skills	Assignment 1: 50	
AJAX implementation REST API		47

A nice example of a functional REST API. Good example of REST endpoints, some evidence of testing. Good use of Express for memory / server. Really good and informative readme.md PATCH request didn't look to work based on documentation that was available. Some nice little touches with the ascii endpoint and the funfacts endpoint. Is that true about the Austin Princess?! 😊 I preferred the Ambassador VP!		
5. Transferable Skills for Life and Professional Practice	Assignment 1: 5	
Presentation Good presentation. A better defined introduction and conclusion would have been desirable. Word count a little high and could have been avoided by removing repetition. Figures should have captions numbering them.		2

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
