

ACCA6024 - Cloud Based Services JE

by Jacob Evans

Submission date: 08-Dec-2025 06:33PM (UTC+0000)

Submission ID: 270080835

File name: 50439_Jacob_Evans_ACCA6024_-_Cloud_Based_Services_JE_708807_1022637012.docx (1.33M)

Word count: 1658

Character count: 8232



ACCA6024 - CLOUD BASED WEB SERVICES

Assignment

Design, develop, test and deploy a web application in a
cloud environment

Jacob Evans
2311160@student.uwtsd.ac.uk

Contents

Part 1	2
REST API	2
GRAPHQL	3
WebSocket	4
Part 2	5
DELETE	7
PUT	8
PATCH	9
PUBLISH	9
Conclusion	12
References	13
Appendix - Copy of Updated Code	14
Test.http	14
Index.js	15
Index.html	17
script.js.....	18

Part 1

REST API

A “Rest” API, also known as “RESTful” that allows for content, media, algorithms, data and other digital resources available through web URLs [1]. This is the most commonly used API and are under the constraints of using HTTP methods such as GET, POST, PUT, PATCH, DELETE. REST uses the JSON, XML, HTML formats to acquire information, and are best for using web and mobile applications. It “rides” on the web’s foundation.

The REST can only be called a “REST” API, it must respect the constraints given to it, such as;

- UI (Uniform Interface)
- Client Server Separation
- Stateless Operations
- Cacheable responses
- Layered system

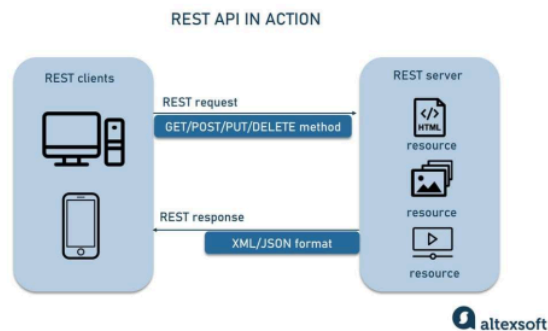


FIGURE 1 - REST API [1]

The REST API works initially through a client request. The application makes an HTTP request to an API end point, which then travels over the network to reach the server it wants to reach. The server processes the data and can question or update the server database. A response is generated, whether it be a client error or server error (status codes such as 400 Bad Request and 404 Not Found), informational status codes (100 Continue, 101 Switching Protocol) or successful status codes (200 OK) [1].

As stated previously, the REST API follows the GET, POST, PUT, PATCH, DELETE HTTP methods.

GET retrieves data only. It can only read and relay the data from the server, while POST is where data can be created or submitted. POST allows for changes to be made on the server, GET cannot. GET is useful for fetching the data created by POST.

PUT allows for the data to be updated. This is incredibly useful for databases such as music libraries, shopping carts etc, as this just updates the information already provided by the server resource. An example of this would be rating music from a music library, where rather than attempting to POST a song review that had been posted previously, PUT allows for the review to be updated easily and efficiently. **PATCH** is similar, whereas you only want to update one field/feature of the data.

DELETE is simply a request to delete the resource. It will be removed from the collection of resources.

GRAPHQL

This API is a query language for API's that enables the client to interact with a single endpoint with only the data they have requested. This improves the performance, as GraphQL does not return the data the client didn't ask for, which in turn reduces the data that is being transferred. This is different from REST, as REST does not allow the client to "dictate the shape of the responses". GraphQL is written in SDL (Schema Definition Language). The request is written specifically, and the response is then "mirrored", so the client obtains only the information they have written GraphQL for specifically.

Unlike GET and POST, GraphQL performs the update to the data and immediately returns the requested data back to the client [2], rather than having to request a "POST", and to then request a "GET". GraphQL has one request and response, while the combination of "POST" and "GET" have 2 requests and responses (GraphQL twice as fast for requesting data). Despite this, GraphQL often returns a 200 OK response, even if there is an error.

Aspect	REST	GraphQL
Endpoints	Multiple resource endpoints	Single /graphql endpoint
Data fetching	Server defines response shape	Client defines response shape
Over-fetching	Common	None; client requests only needed fields
Versioning	Managed via URL (/v1, /v2)	Evolved via schema; no breaking changes
Caching	Built into HTTP	Custom or layer-specific caching
Real-time data	Requires separate setup	Built-in with subscriptions

FIGURE 2 - REST vs GraphQL [2]

Due to the defined data request, GraphQL is optimised for mobile clients, as it only pulls the required/recommended data, which lowers latency and improves the client experience.

REST and GraphQL compliment each other.

WebSocket

WebSocket is a communication protocol, that stays open, allowing for real-time bidirectional communication. The HTTP is initialised with a “handshake” [3], which allows the client and server to send messages anytime. This is all over a single TCP connection. This allows for the client and the server to communicate instantaneously, which is perfect for applications requiring real-time updates.

A great example of this API being used is within the WhatsApp application, as real-time communication is used. WhatsApp also allows the user to see when the

REST is one way, in comparison to WebSocket, as the client requests information, and the server then responds. The server cannot push updates or requests automatically, as they require the initial request as seen in figure 1. This also means that the REST does not keep a continuous connection, and opens/reuses connections, which are considered separate events. This makes REST a stateless system, as it generally does not store the client information between requests.

WebSocket would be stateful, as it is constantly open until the handshake is closed, therefore the system remembers information about the client between requests. GraphQL is also stateless due to not storing any client information between requests.

Feature	REST	SOAP	GraphQL	gRPC	WebSocket
Protocol	HTTP/HTTPS	HTTP, SMTP, TCP	HTTP/HTTPS	HTTP/2	TCP/WebSocket
Data format	JSON, XML, HTML	XML only	JSON	Protocol Buffers (binary)	Binary, Text, JSON
Learning curve	Low	High	Medium	Medium	Low
Performance	Good	Moderate	Good	Excellent	Excellent
Browser support	Full	Full	Full	Limited (needs gRPC-Web)	Full
Caching	Native HTTP caching	Custom implementation	Custom strategies	Custom implementation	Not applicable
Real-time support	No (polling required)	No	Yes (via subscriptions)	Yes (streaming)	Yes (native)
State management	Stateless	Can be stateful	Stateless	Stateless	Stateful (persistent)
Security	HTTPS, OAuth, API keys	WS-Security (signing, encryption), TLS	Custom, OAuth	TLS, custom auth	TLS, tokens
Versioning	URL or header based	WSDL versioning	Schema evolution	Protobuf versioning	Custom
Error handling	HTTP status codes	SOAP faults	GraphQL errors	Status codes	Custom messages
Best for	General web services	Regulated industries	Client-driven apps	High-performance systems	Real-time applications

FIGURE 3 - DIFFERENCES BETWEEN APIS [3]

Part 2

This is an example of both “POST” and “GET”. The POST shows that there is an input for a title called “Just Pretend” with other information (by the artist and rating).

```
{  
  "title": "Just Pretend",  
  "artist": "Bad Omens",  
  "rating": 5  
}
```

This is showcasing that the POST has worked - that the song has been created, and “POSTed” to the html website.

6	Just Pretend	Bad Omens	5	draft
---	--------------	-----------	---	-------

By clicking “GET”, the information from the html/website has returned the information from the website, with “ID” number.

```
{  
  "id": 6,  
  "title": "Just Pretend",  
  "artist": "Bad Omens",  
  "rating": 5,  
  "status": "draft"  
}  
]
```

The same can be done through the HTML. By inputting the song details, and clicking “POST”, the information has been created at the bottom of the table under ID 7.

Music Library (REST API)

Song Details

ID

Title

Artist

Rating

[GET all songs](#) [GET by ID](#) [POST create](#) [PUT update](#) [PATCH rating](#) [Publish \(workflow\)](#) [DELETE](#)

Songs in API

ID	Title	Artist	Rating	Status
1	Caramel	Sleep Token	5	draft
2	All Star	Smash Mouth	5	draft
4	Smoke W**d Everyday	Snoop Dogg	3	published
5	Blinding Lights	The Weeknd	5	draft
6	Just Pretend	Bad Omens	5	draft
7	GGGG	GGGGG	1	draft

When “GET” is done in WSC, the whole database and new addition is returned.

```
"status": "draft"
},
{
  "id": 6,
  "title": "Just Pretend",
  "artist": "Bad Omens",
  "rating": 5,
  "status": "draft"
},
{
  "id": 7,
  "title": "GGGG",
  "artist": "GGGGG",
  "rating": 1,
  "status": "draft"
}
]
```

DELETE

Here's an example of DELETE in RESTful. We can see down below that there is a song under ID 7 requested to be deleted (this is GGGGG, can be seen as an error). When loading up the API, website, there is no song under ID 7. When attempting retrieve the same song under ID 7, we can see an error message stating "Song not found", suggesting that the delete was successful.

```
test.http > ...
Send Request
POST http://localhost:3000/songs
Content-Type: application/json

{
  "title": "Blinding Lights",
  "artist": "The Weeknd",
  "rating": 5,

  "title": "Just Pretend",
  "artist": "Bad Omens",
  "rating": 5
}

### Get all songs
Send Request
GET http://localhost:3000/songs

### DELETE a song (change the ID to one that exists)
Send Request
DELETE http://localhost:3000/songs/7
```

```
1 HTTP/1.1 204 No Content
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Date: Mon, 08 Dec 2025 15:48:04 GMT
5 Connection: close
6
7
```

Songs in API				
ID	Title	Artist	Rating	Status
1	Caramel	Sleep Token	5	draft
2	All Star	Smash Mouth	5	draft
4	Smoke W*d Everyday	Snoop Dogg	3	published
5	Blinding Lights	The Weeknd	5	draft
6	Just Pretend	Bad Omens	5	draft

```
{
  "title": "Blinding Lights",
  "artist": "The Weeknd",
  "rating": 5,

  "title": "Just Pretend",
  "artist": "Bad Omens",
  "rating": 5
}

### Get all songs
Send Request
GET http://localhost:3000/songs

### Get On
Send Request Follow link (ctrl + click)
GET http://localhost:3000/songs/7

### DELETE a song (change the ID to one that exists)
Send Request
DELETE http://localhost:3000/songs/7
```

```
1 X-Powered-By: Express
2 Access-Control-Allow-Origin: *
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 28
5 ETag: W/"1c-608kn29sdBxqMICv4r5/Fr8NKig"
6 Date: Mon, 08 Dec 2025 15:50:22 GMT
7 Connection: close
8
9
10 {
11   "message": "Song not found"
12 }
```

PUT

This is an example of “PUT” being used to update something on the website/database. From previous screenshots, we can see that song ID 1 was “Caramel”. Since using the PUT command, we can see that the song has been updated to “Old Title”, and “Unknown Artist” as requested.

```
### UPDATE A SONG
Send Request
PUT http://localhost:3000/songs/1
Content-Type: application/json

{
  "title": "Old Title",
  "artist": "Unknown Artist",
  "rating": 3
}
```

```
10 {
11   "id": 1,
12   "title": "Old Title",
13   "artist": "Unknown Artist",
14   "rating": 3,
15   "status": "draft"
16 }
```

Music Library (REST API)

Song Details

ID

For update/delete/publish

Title

Song title

Artist

Artist name

Rating

1-5

GET all songs

GET by ID

POST create

PUT update

PATCH rating

Publish (workflow)

DELETE

Songs in API

ID	Title	Artist	Rating	Status
1	Old Title	Unknown Artist	3	draft
2	All Star	Smash Mouth	5	draft
4	Smoke W**d Everyday	Snoop Dogg	3	published
5	Blinding Lights	The Weeknd	5	draft
6	Just Pretend	Bad Omens	5	draft

PATCH

This is an example of PATCH being used. We see that the initial rating for song ID 1 was 3. By using the PATCH method, and identifying which feature to edit, we can see that the edit it has been made. Song ID 1 has now had it's rating increased to 4.2, rather than the previous 3.

```
4  ### Update One single thing on the data
5  Send Request
6  PATCH http://localhost:3000/songs/1
7  Content-Type: application/json
8
9  {
10   "rating": 4.2
11 }
12
13 10 ~ [
14 11 ~ {
15 12   "id": 1,
16 13   "title": "Old Title",
17 14   "artist": "Unknown Artist",
18 15   "rating": 4.2,
19 16   "status": "draft"
20 17 },
21 18 ~ {
```

Songs in API

ID	Title	Artist	Rating	Status
1	Old Title	Unknown Artist	4.2	draft
2	All Star	Smash Mouth	5	draft
4	Smoke W**d Everyday	Snoop Dogg	3	published
5	Blinding Lights	The Weeknd	5	draft
6	Just Pretend	Bad Omens	5	draft

PUBLISH

This is the PUBLISH method. Once this is POSTed, this cannot be returned to draft without deleting that whole input. We can see in the screenshots below that ID 1 has now had it's status changed to "published". When trying to POST it back to draft, it brings up an error.

```
1  {
2   "rating": 4.2
3 }
4
5 6  ### Publish a song
7  Send Request
8  POST http://localhost:3000/songs/1/publish
9
10 10 ~ {
11 11   "id": 1,
12 12   "title": "Old Title",
13 13   "artist": "Unknown Artist",
14 14   "rating": 4.2,
15 15   "status": "published"
16 16 }
```

Songs in API

ID	Title	Artist	Rating	Status
1	Old Title	Unknown Artist	4.2	published
2	All Star	Smash Mouth	3	draft
4	Smoke W**d Everyday	Snoop Dogg	3	published
5	Blinding Lights	The Weeknd	5	draft
6	Just Pretend	Bad Omens	5	draft

```
Send Request
{
  "rating": 4.2
}

### Publish a song
Send Request
POST http://localhost:3000/songs/1/draft

11 <!DOCTYPE html>
12 <html lang="en">
13 <head>
14 <meta charset="utf-8">
15 <title>Error</title>
16 </head>
17 <body>
18 <pre>Cannot POST /songs/1/draft</pre>
19 </body>
20 </html>
21
```

We also cannot republish an already published song, as seen below, showcasing an invalid workflow transition, as the song is already published.

```
Send Request
PATCH http://localhost:3000/songs/1
Content-Type: application/json

{
  "rating": 4.2
}

### Publish a song
Send Request
POST http://localhost:3000/songs/1/publish

7 Date: Mon, 08 Dec 2025 16:35:13 GMT
8 Connection: close
9
10 {
11   "message": "Invalid workflow transition: published + published"
12 }
```

We can also see from this screenshot that an error code is produced when an unknown song ID is attempted to be published; as there is no song under ID 10, this cannot be published.

```
31 PATCH http://localhost:3000/songs/1
32 {
33   "rating": 4.2
34 }
35
36 ### Publish a song
37 Send Request
38 POST http://localhost:3000/songs/10/publish

1 HTTP/1.1 404 Not Found
2 X-Powered-By: Express
3 Access-Control-Allow-Origin: *
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 28
6 ETag: W/"1c-608kn29sdbxqM1Cv4r5/rFbUkig"
7 Date: Mon, 08 Dec 2025 16:39:09 GMT
8 Connection: close
9
10 {
11   "message": "Song not found"
12 }
```

Test Title:	<i>Launching API</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>API is launched, and a https://localhost is created for the music API to be accessed.</i>
Expected Outcome:	<i>"Music API running on http://localhost:3000" should be seen in the terminal, as well as clicking on the link allows the user to open the website</i>

Test Title:	<i>Creating Song (through POST)</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>A song can be posted whether in the Visual Studio Code Rest Client (test.https) or on the website itself</i>
Expected Outcome:	<i>When a song is POST through the website, the database seen below the buttons adds a new data feature based on the input through the website.</i> <i>When a song is POST through the VSC test.https, the information should be added to the website after refreshing.</i> <i>HTTP Status Code 201 Created expected</i>

Test Title:	<i>GET All Songs</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>Ensures that all songs posted via VSC test or website are able to be retrieved through "GET".</i>
Expected Outcome:	<i>JSON array returned, including the new created songs. HTTP Status Code 200 OK expected</i>
Test Title:	<i>Get Song from Specific ID</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>Ensures that the specific song ID is retrieved from the VSC test or website.</i>
Expected Outcome:	<i>JSON array returned, including the new created songs. HTTP Status Code 200 OK expected</i>

Test Title:	<i>Invalid Re-Publish</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>Attempt to publish song from ID again.</i>
Expected Outcome:	<i>JSON array error returned, "Invalid workflow transition" HTTP Status Code 400 expected</i>

Test Title:	<i>Delete Song</i>
Test Author:	<i>J. Evans</i>
Test Description:	<i>Song is deleted through specific ID number</i>
Expected Outcome:	<i>Song removed on website and when GET is used. GET used on specific ID, returns JSON array "Song not found" Status code 404 expected</i>

Conclusion

Through research and application of REST, we can see just how useful this API is. It's simple to use and learn and works efficiently and effectively for web applications. We have to understand that there is no one single API that should be used, as all different types of APIs can work hand in hand with each other.

REST has its limitations in areas such as real-time communication, as well as it may not be as secure compared to another API like SOAP, so we can only assume that developers will pick their recommended API of choice for the task they have been provided.

References

- [1] “Rest API Examples”, POSTMAN TEAM,” <https://blog.postman.com/rest-api-examples/>”
- [2] “What is GraphQL”, POSTMAN TEAM,” <https://blog.postman.com/what-is-a-graphql-api-how-does-it-work/>”
- [3] “Different Types of APIs”, POSTMAN TEAM, “<https://blog.postman.com/different-types-of-apis/>”

Appendix - Copy of Updated Code

Test.http

```
testhttp > GET /songs
1  ### Create a song
   Send Request
2  POST http://localhost:3000/songs
3  Content-Type: application/json
4
5  {
6    "title": "Blinding Lights",
7    "artist": "The Weeknd",
8    "rating": 5
9  }
10 ### Get all songs
   Send Request
11 GET http://localhost:3000/songs
12
13 ### Get One song
   Send Request
14 GET http://localhost:3000/songs/7
15
16 ### DELETE a song (change the ID to one that exists)
   Send Request
17 DELETE http://localhost:3000/songs/7
18
19 ### UPDATE A SONG
   Send Request
21 PUT http://localhost:3000/songs/1
22 Content-Type: application/json
23
24 {
25   "title": "Old Title",
26   "artist": "Unknown Artist",
27   "rating": 3
28 }
29
30 ### Update One single thing on the data
   Send Request
31 PATCH http://localhost:3000/songs/1
32 Content-Type: application/json
33
34 {
35   "rating": 4.2
36 }
37
38 ### Publish a song
   Send Request
39 POST http://localhost:3000/songs/10/publish
```

Index.js

```
# index.js > ...
1  const express = require("express");
2  const cors = require("cors");
3
4  const app = express();
5  app.use(cors());
6  app.use(express.json());
7
8  // In-memory "database"
9  let nextId = 1;
10 const songs = []; // each song: {id, title, artist, rating, status}
11
12 // Workflow statuses
13 const validStatuses = ["draft", "published"];
14
15 function validateTransition(current, newStatus) {
16   if (!validStatuses.includes(newStatus)) {
17     return "Invalid status value";
18   }
19   if (current === "draft" && newStatus === "published") return null;
20   return `Invalid workflow transition: ${current} -> ${newStatus}`;
21 }
22
23 // ===== ROUTES =====
24
25 // GET all songs
26 app.get("/songs", (req, res) => {
27   res.json(songs);
28 });
29
30 // GET one song by id
31 app.get("/songs/:id", (req, res) => {
32   const song = songs.find(s => s.id == req.params.id);
33   if (!song) return res.status(404).json({ message: "Song not found" });
34   res.json(song);
35 });
36
37 // POST create a new song (status always starts as "draft")
38 app.post("/songs", (req, res) => {
39   const { title, artist, rating } = req.body;
40   if (!title || !artist || !rating) {
41     return res.status(400).json({ message: "title, artist and rating are required" });
42   }
43
44   const song = {
45     id: nextId++,
46     title,
47     artist,
48     rating,
49     status: "draft"
50   };
51   songs.push(song);
52   res.status(201).json(song);
53 });
54
55 // PUT full update (replace title/artist/rating, keep status)
56 app.put("/songs/:id", (req, res) => {
57   const song = songs.find(s => s.id == req.params.id);
58   if (!song) return res.status(404).json({ message: "Song not found" });
59
60   const { title, artist, rating } = req.body;
61   if (!title || !artist || !rating) {
62     return res.status(400).json({ message: "title, artist and rating are required" });
63   }
64
65   song.title = title;
66   song.artist = artist;
67   song.rating = rating;
68
69   res.json(song);
70 });
```

```

// PATCH partial update
app.patch("/songs/:id", (req, res) => {
  const song = songs.find(s => s.id == req.params.id);
  if (!song) return res.status(404).json({ message: "Song not found" });

  const { title, artist, rating, status } = req.body;

  if (status !== undefined) {
    const error = validateTransition(song.status, status);
    if (error) return res.status(400).json({ message: error });
    song.status = status;
  }

  if (title !== undefined) song.title = title;
  if (artist !== undefined) song.artist = artist;
  if (rating !== undefined) song.rating = rating;

  res.json(song);
});

// DELETE a song
app.delete("/songs/:id", (req, res) => {
  const index = songs.findIndex(s => s.id == req.params.id);
  if (index === -1) return res.status(404).json({ message: "Song not found" });

  songs.splice(index, 1);
  res.status(204).send();
});

// WORKFLOW: publish a song (draft -> published)
app.post("/songs/:id/publish", (req, res) => {
  const song = songs.find(s => s.id == req.params.id);
  if (!song) return res.status(404).json({ message: "Song not found" });

  const error = validateTransition(song.status, "published");
  if (error) return res.status(400).json({ message: error });

  song.status = "published";
  res.json(song);
});

// Start server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Music API running on http://localhost:${PORT}`);
});

```

Index.html

```
index.html > html > body > div.container > section > div
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8" />
5   <title>Music Library Client</title>
6   <link rel="stylesheet" href="style.css" />
7 </head>
8 <body>
9   <div class="container">
10     <h1>Music Library (REST API)</h1>
11
12     <section>
13       <h2>Song Details</h2>
14
15       <div class="form-row">
16         <label for="song-id">ID</label>
17         <input type="number" id="song-id" placeholder="For update/delete/publish" />
18       </div>
19
20       <div class="form-row">
21         <label for="title">Title</label>
22         <input type="text" id="title" placeholder="Song title" />
23       </div>
24
25       <div class="form-row">
26         <label for="artist">Artist</label>
27         <input type="text" id="artist" placeholder="Artist name" />
28       </div>
29
30       <div class="form-row">
31         <label for="rating">Rating</label>
32         <input type="number" id="rating" min="1" max="5" step="0.1" placeholder="1.0" />
33       </div>
34
35       <div>
36         <button class="primary" id="btn-load">GET all songs</button>
37         <button class="primary" id="btn-get-one">GET by ID</button>
38         <button class="secondary" id="btn-create">POST create</button>
39         <button class="secondary" id="btn-update">PUT update</button>
40         <button class="secondary" id="btn-patch">PATCH rating</button>
41         <button class="info" id="btn-publish">Publish (workflow)</button>
42         <button class="danger" id="btn-delete">DELETE</button>
43       </div>
44     </section>
45
46     <div id="message"></div>
47
48     <section>
49       <h2>Songs in API</h2>
50       <table id="songs-table">
51         <thead>
52           <tr>
53             <th>ID</th>
54             <th>Title</th>
55             <th>Artist</th>
56             <th>Rating</th>
57             <th>Status</th>
58           </tr>
59         </thead>
60         <tbody id="songs-body">
61           <!-- JS will fill this -->
62         </tbody>
63       </table>
64     </section>
65   </div>
66
67   <script src="script.js"></script>
68 </body>
69 </html>
70
```

script.js

```
index.js  index.html  JS script.js  X  {} package.json  test.http
script.js > [9] tbody
const API_BASE = "http://localhost:3000";

// Inputs
const idInput = document.getElementById("song-id");
const titleInput = document.getElementById("title");
const artistInput = document.getElementById("artist");
const ratingInput = document.getElementById("rating");

// UI elements
const msgBox = document.getElementById("message");
const tbody = document.getElementById("songs-body");

function showMessage(text, type = "success") {
  msgBox.textContent = text;
  msgBox.className = type === "error" ? "error" : "success";
  msgBox.style.display = "block";
}

function clearMessage() {
  msgBox.textContent = "";
  msgBox.className = "";
  msgBox.style.display = "none";
}

function renderSongs(songs) {
  tbody.innerHTML = "";
  songs.forEach(song => {
    const tr = document.createElement("tr");
    tr.innerHTML = `
      <td>${song.id}</td>
      <td>${song.title}</td>
      <td>${song.artist}</td>
      <td>${song.rating}</td>
      <td>${song.status}</td>
    `;
    tbody.appendChild(tr);
  });
}

/* ---- API CALLS ---- */

async function getAllSongs() {
  clearMessage();
  const res = await fetch(`${API_BASE}/songs`);
  const data = await res.json();
  renderSongs(data);
}

async function getSongById() {
  clearMessage();
  const id = idInput.value;
  if (!id) return showMessage("Enter an ID for GET by ID", "error");

  const res = await fetch(`${API_BASE}/songs/${id}`);
  if (!res.ok) return showMessage("Song not found", "error");
  const song = await res.json();
  renderSongs([song]);
}

async function createSong() {
  clearMessage();
  const body = {
    title: titleInput.value,
    artist: artistInput.value,
    rating: Number(ratingInput.value)
  };

  const res = await fetch(`${API_BASE}/songs`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(body)
  });

  if (!res.ok) return showMessage("Failed to create song", "error");
  showMessage("Song created");
  getAllSongs();
}
```

```

const body = {
  title: titleInput.value,
  artist: artistInput.value,
  rating: Number(ratingInput.value)
};

const res = await fetch(`${API_BASE}/songs/${id}`, {
  method: "PUT",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(body)
});

if (!res.ok) return showMessage("PUT update failed", "error");
showMessage("Song updated (PUT)");
getAllSongs();
}

async function patchSongRating() {
  clearMessage();
  const id = idInput.value;
  if (!id) return showMessage("Enter ID for PATCH", "error");

  const body = { rating: Number(ratingInput.value) };

  const res = await fetch(`${API_BASE}/songs/${id}`, {
    method: "PATCH",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(body)
  });

  if (!res.ok) return showMessage("PATCH failed", "error");
  showMessage("Rating updated (PATCH)");
  getAllSongs();
}

async function deleteSong() {
  clearMessage();
  const id = idInput.value;
  if (!id) return showMessage("Enter ID to delete", "error");

  const res = await fetch(`${API_BASE}/songs/${id}`, {
    method: "DELETE"
  });

  if (!res.ok) return showMessage("Delete failed", "error");
  showMessage("Song deleted");
  getAllSongs();
}

async function publishSong() {
  clearMessage();
  const id = idInput.value;
  if (!id) return showMessage("Enter ID to publish", "error");

  const res = await fetch(`${API_BASE}/songs/${id}/publish`, {
    method: "POST"
  });

  if (!res.ok) {
    const err = await res.json().catch(() => ({}));
    return showMessage(err.message || "Publish failed", "error");
  }

  showMessage("Song published (draft -> published)");
  getAllSongs();
}

/* ---- Button event handlers ---- */

document.getElementById("btn-load").addEventListener("click", getAllSongs);
document.getElementById("btn-get-one").addEventListener("click", getSongById);
document.getElementById("btn-create").addEventListener("click", createSong);
document.getElementById("btn-update").addEventListener("click", updateSongPut);
document.getElementById("btn-patch").addEventListener("click", patchSongRating);
document.getElementById("btn-delete").addEventListener("click", deleteSong);
document.getElementById("btn-publish").addEventListener("click", publishSong);

// Load songs when page opens
getAllSongs();

```

FINAL GRADE

GENERAL COMMENTS

67 /100

	Marks available	Marks awarded
Generic Assessment Criteria		
1. Engagement with Literature Skills	Assignment 1: 5	
A very low number of sources. Some academic and credible sources would have been desirable.		1
2. Knowledge and Understanding Skills	Assignment 1: 10	
Investigation, use-cases and advantages / disadvantages. Reasonable knowledge demonstrated of REST, GraphQL and web sockets. Some reasonable discussion of the workings. Possibly more literature sources may have been advantageous in discussing a		6

little more detail here. A more detailed discussion of comparisons and use-cases would have been beneficial, although a nice comparison table for the different APIs shown.		
1. Cognitive and Intellectual Skills	Assignment 1: 30	
Design and documentation Test documentation and testing Good use of test tables, although further boxes for PASS or FAIL and actual outcome would have been a desirable addition. Some reasonable discussion of the api and the endpoints.		18
4. Practical Application Skills	Assignment 1: 50	
AJAX implementation REST API A nice example of a functional REST API. Commented index.js is good. A well packaged folder with package.json. A readme file may have been advantageous.		40

5. Transferable Skills for Life and Professional Practice	Assignment 1: 5	
Presentation Reasonable presentation, an introduction would have been desirable. All images / figures should have been captioned.		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

