REST API Guide

This guide teaches you how to interact with REST APIs using JavaScript, covering the essential techniques needed to solve the REST station challenges.

Table of Contents

Basic HTTP Requests

What is a REST API?

A REST API (Representational State Transfer) is a way for applications to communicate over HTTP. You send requests to specific URLs (endpoints) and receive data back, usually in JSON format.

Making GET Requests with `fetch()`

The `fetch()` function is the modern way to make HTTP requests in JavaScript:

```
// Basic GET request
const response = await fetch('https://api.example.com/data');
const data = await response.json();
console.log(data);
```

Key concepts:

- `fetch()` returns a Promise, so use `await` to wait for the response
- `.json()` parses the response body as JSON (also returns a Promise)
- Always check if the request was successful before using the data

Checking Response Status

```
const response = await fetch('https://api.example.com/data');

if (!response.ok) {
  console.error(`HTTP Error: ${response.status}`);
  throw new Error(`Request failed: ${response.statusText}`);
}

const data = await response.json();
```

API Authentication

Many APIs require authentication to verify your identity and permissions.

Header-Based Authentication

Some APIs use custom headers for authentication:

```javascript
const response = await fetch('https://api.example.com/protected', {

  headers: {

    'X-API-Key': 'your-secret-key-here',

    'Content-Type': 'application/json'

  }

});



const data = await response.json();
```

POST Requests with JSON Body

Some endpoints require you to send data in the request body:

```javascript
const response = await fetch('https://api.example.com/authenticate', {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json'

  },

  body: JSON.stringify({

    username: 'agent007',

    clearance: 'omega'

  })

});



const data = await response.json();

console.log(data.api_key); // Extract API key from response
```

Key concepts:

- Use `method: 'POST'` for POST requests
- Set `Content-Type: application/json` when sending JSON
- Use `JSON.stringify()` to convert JavaScript objects to JSON strings

Pagination

When APIs have large datasets, they split the data into pages to avoid overwhelming responses.

Understanding Pagination

Pagination typically works like this:

- Request page 1, get first 10 items
- Request page 2, get next 10 items
- Continue until you reach the last page

Basic Pagination Loop

```
const uuid = 'abc-123-def-456';

const baseUrl = `https://api.example.com/data/${uuid}`;


// Get first page to find total pages

const firstPageResponse = await fetch(`${baseUrl}?page=1`);

const firstPageData = await firstPageResponse.json();

const totalPages = firstPageData.pagination.total_pages;


console.log(`Total pages: ${totalPages}`);


// Loop through all pages

for (let page = 1; page <= totalPages; page++) {

  const response = await fetch(`${baseUrl}?page=${page}`);

  const data = await response.json();


  console.log(`Page ${page}/${totalPages}:`, data.entries);


  // Process each entry on this page

  for (const entry of data.entries) {

    // Do something with entry

    console.log(entry);

  }

}
```

Advanced: Processing Multiple Resources

When you need to paginate through multiple different resources:

```
const suspects = ['uuid-1', 'uuid-2', 'uuid-3'];


for (const uuid of suspects) {
  console.log(`\nProcessing suspect: ${uuid}`);


  // Get info to find page count

  const infoResponse = await fetch(`/api/surveillance/${uuid}?page=1`);

  const info = await infoResponse.json();

  const totalPages = info.pagination.total_pages;


  // Loop through all pages for this suspect

  for (let page = 1; page <= totalPages; page++) {

    const response = await fetch(`/api/surveillance/${uuid}?page=${page}`);

    const pageData = await response.json();


    // Search for specific data

    for (const entry of pageData.data) {

      if (entry.important === true) {

        console.log(`Found important data on page ${page}!`);

        console.log(entry);

      }

    }

  }

}
```

Base64 Encoding/Decoding

Base64 is a way to encode binary data as ASCII text. APIs often use it to transmit data that might contain special characters.

Decoding Base64 in JavaScript

```
// Base64 string from API

const encodedString = "eyJ3ZWFwb24iOiJQaGFzZXIiLCJ0eXBlIjoiRW5lcmd5In0=";


// Decode to regular string

const decodedString = atob(encodedString);

console.log(decodedString); // {"weapon":"Phaser","type":"Energy"}


// Parse as JSON

const jsonObject = JSON.parse(decodedString);
```

```
console.log(jsonObject.weapon); // "Phaser"
```

Key functions:

- `atob(base64String)` - Decodes Base64 to string ("ascii to binary")
- `btoa(string)` - Encodes string to Base64 ("binary to ascii")

Common Pattern: API Returns Base64 Data

```
const response = await fetch('/api/surveillance/uuid?page=1');

const data = await response.json();


// API returns: { data: "base64string...", encoding: "base64" }

const encodedData = data.data;


// Decode the Base64 string

const decodedJson = atob(encodedData);


// Parse as JSON to get array of entries

const entries = JSON.parse(decodedJson);


console.log(entries); // Array of surveillance entries
```

Practical Examples

Example 1: Get API Key from Authentication Endpoint

```
// Step 1: POST credentials to get API key

const authResponse = await fetch('/api/v1/spider/key', {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json'

  },

  body: JSON.stringify({

    agent_id: 'investigation_team',

    clearance: 'omega'

  })

});
```

```
const authData = await authResponse.json();

const apiKey = authData.key;


console.log('Got API key:', apiKey);


// Step 2: Use API key in subsequent requests

const protectedResponse = await fetch('/api/v1/spider/agents', {

  headers: {

    'X-SPIDER-Key': apiKey

  }

});


const agents = await protectedResponse.json();

console.log('Agents:', agents);
```

Tips & Best Practices

1. **Always use `await`** - Don't forget to await both `fetch()` and `.json()`
2. **Check response status** - Use `response.ok` or check `response.status`
3. **Use descriptive variable names** - `pageData`, `decodedEntries`, etc.
4. **Log progress** - Use `console.log()` to track your script's progress
5. **Handle errors** - Wrap fetch calls in try/catch blocks
6. **Read API documentation** - Check `/info` endpoints to understand the API
7. **Test incrementally** - Start with one request, then add pagination/loops

---

Common Pitfalls

❌ **Forgetting to await**

```
const data = fetch('/api/data').json(); // Missing await!
```

✅ **Correct**

```
const response = await fetch('/api/data');

const data = await response.json();
```

❌ **Not checking response status**

```
const response = await fetch('/api/data');

const data = await response.json(); // Might fail if response is error
```

✅ **Correct**

```
const response = await fetch('/api/data');

if (!response.ok) {

  throw new Error(`HTTP error: ${response.status}`);

}

const data = await response.json();
```

❌ **Forgetting to decode Base64**

```
console.log(data.data); // Prints Base64 gibberish
```

✅ **Correct**

```
const decoded = JSON.parse(atob(data.data));

console.log(decoded); // Prints actual data
```

Quick Reference

```
// GET request

const response = await fetch('/api/endpoint');

const data = await response.json();


// POST request with JSON

const response = await fetch('/api/endpoint', {

  method: 'POST',

  headers: { 'Content-Type': 'application/json' },

  body: JSON.stringify({ key: 'value' })

});


// Custom headers

const response = await fetch('/api/endpoint', {

  headers: { 'X-Custom-Header': 'value' }

});


// Query parameters
```

```
const url = `/api/endpoint?page=${pageNum}&filter=${filterValue}`;



// Decode Base64

const decoded = atob(base64String);



// Parse JSON

const object = JSON.parse(jsonString);
```

Resources

- MDN: Fetch API
- MDN: Using Fetch
- MDN: atob()
- JSON.parse() documentation