

API Service Setup

what is this Axios

Great question, Prasun! Let's **break down what Axios actually is** — and I'll explain it in a simple yet developer-friendly way 🙌












◆ What is Axios?

Axios is a **promise-based HTTP client** for the browser and Node.js used to **make HTTP requests** (like `GET`, `POST`, `PUT`, `DELETE`) from the frontend to the backend or any external API.

Think of it as the tool that helps your web app **talk to your server** or **fetch/send data** from remote APIs.

◆ Why Use Axios (Instead of fetch)?

Axios vs. `fetch()` (native in browsers):

Feature	Axios 	fetch  / 
Simpler syntax	Yes	Slightly verbose
Auto JSON parsing	 <code>.data</code> contains parsed data	 need <code>res.json()</code> manually
Interceptors (req/res)	 Powerful support	 Not built-in
Global config	 Easy to configure baseURL etc.	 Manual
Error handling	 Clean and consistent	 More boilerplate

◆ Where is Axios Used in Projects?

You typically use it in:

- **Frontend React apps:** to call your Express, Django, or any backend API
 - **Vue/Angular apps:** same purpose
 - **Node.js servers:** to call other APIs/services
 - **Full stack apps:** part of the client-to-server communication
-

◆ Axios Code Example

👉 Simple GET Request:

js

```
import axios from 'axios';

axios.get('https://api.example.com/users')
  .then(response => {
    console.log(response.data); // automatically parsed JSON
  })
  .catch(error => {
    console.error(error);
  });
```

👉 Simple POST Request:

js

```
axios.post('https://api.example.com/login', {
  username: 'prasun',
  password: '123456'
})
  .then(res => console.log(res.data))
  .catch(err => console.error(err));
```

◆ Advanced Features Axios Supports:

- Creating an **Axios instance** with custom headers, baseUrl, and timeout.
 - Adding **Request Interceptors** (add token to every request).
 - Adding **Response Interceptors** (handle 401 errors globally).
 - **Timeouts, Cancel tokens, Progress tracking**, etc.
-

◆ In Your Case (Expense Tracker Project)

You used Axios to:

- Call login and register APIs
- Add, delete, download incomes/expenses
- Attach the token in every request using request interceptors
- Handle errors using response interceptors

Perfect real-world usage! *100*

Let me know if you'd like a **mind map, comparison with fetch, or real-world code snippets**.
I can build one for you!

Complete List of all the API like FetchAPI to send request to server

Absolutely, Prasun! Here's a **complete list of APIs/tools/libraries** you can use to **send HTTP requests to a server** (like you do with Axios or Fetch API). Each has different strengths and is suited to different environments or preferences.



Complete List of APIs & Libraries for Making HTTP Requests

♦ 1. Browser-Based (Frontend) APIs/Libraries

Name	Description
Fetch API	Built-in modern browser API. Uses Promises. Replaces <code>XMLHttpRequest</code> .
Axios	Most popular third-party HTTP client. Used heavily in React, Vue, etc.
XMLHttpRequest	Older, callback-based browser API. Predecessor to <code>fetch</code> .
jQuery.ajax()	AJAX calls in jQuery. Rare now in modern apps.
SuperAgent	Lightweight AJAX library. Easy chaining & plugins.
Ky	Tiny wrapper around <code>fetch</code> with simpler syntax and defaults.
SWR (React)	Data-fetching library using hooks. Built on top of <code>fetch</code> or Axios.
React Query	Powerful data-fetching and caching tool. Not a fetcher by itself, but wraps Axios/fetch.
GraphQL Client Libraries	If you're using GraphQL APIs:

- `Apollo Client` , `urql` , `Relay` etc. |

♦ 2. Node.js/Backend HTTP Clients

Name	Description
Axios	Works in both browser and Node.js.
http/https (Node core)	Built-in Node.js modules for raw HTTP(S) requests.
Got	Modern, promise-based Node.js HTTP client. Cleaner than <code>http</code> .
node-fetch	Fetch API polyfill for Node.js (used when <code>fetch</code> isn't available).

Name	Description
SuperAgent	Also works in Node.js for HTTP requests.
Request (deprecated)	Used to be the most popular, now deprecated.
Needle	Lightweight HTTP client for Node.js.
Wreck	HTTP client built by the Hapi.js team.
GraphQL Clients	Like Apollo Client, urql for server-side GraphQL requests.

◆ 3. Command-Line Tools (for testing)

Tool	Description
cURL	Command-line tool to send HTTP requests. Supports GET, POST, PUT, etc.
HTTPIe	More human-friendly alternative to cURL.
Postman CLI	CLI version of Postman to run API collections.
wget	Command-line utility to download files, make GET requests, etc.

◆ 4. API Testing / Automation Tools

Tool	Description
Postman	Most popular GUI-based API testing tool.
Insomnia	Great Postman alternative, very developer-friendly.
Thunder Client	VS Code extension for testing APIs.




Tool	Description
REST Client	Another VS Code extension — makes HTTP requests from <code>.http</code> files.

◆ 5. Other Popular HTTP Libraries / Framework Integrations

Context	Tool
Angular	<code>HttpClientModule</code> (built-in Angular module for API calls)
Vue	<code>axios</code> , <code>fetch</code> , or <code>vue-resource</code> (deprecated)
React	Use <code>fetch</code> , <code>axios</code> , or libraries like SWR, React Query
Svelte	Uses <code>fetch</code> or <code>axios</code> , or you can roll your own
Flutter	<code>http</code> package, <code>dio</code> for advanced usage
Python	<code>requests</code> , <code>httpx</code> , <code>aiohttp</code>
Java	<code>OkHttp</code> , <code>Retrofit</code> , <code>HttpURLConnection</code>
Golang	<code>net/http</code> , <code>gorequest</code> , <code>resty</code>
Rust	<code>reqwest</code> , <code>hyper</code>
PHP	<code>cURL</code> , <code>Guzzle</code>
C# (.NET)	<code>HttpClient</code> , <code>RestSharp</code>

✓ Summary: What You Should Know

- ✓ In frontend (React/Vue), use **Axios** or **fetch**.
- ✓ In Node.js backend, use **Axios**, **Got**, or **node-fetch**.

-  For GraphQL, use **Apollo Client** or **urql**.
-  For testing, use **Postman**, **Thunder Client**, or **Insomnia**.
-  If you're on CLI, use **curl** or **httpie**.

Would you like a **comparison table** or **flowchart** **YAML** showing when to use what in a full stack app? I can draw that too.

Then what is REST API Graphql and other ?what is this

Great question, Prasun! Let's break down **REST API**, **GraphQL**, and other types of APIs in a simple, clear, and complete way. These are **ways of structuring and designing APIs** for your applications.

1. REST API (Representational State Transfer)

What it is:

REST is the **most common** and traditional way to design APIs using **HTTP methods** (GET, POST, PUT, DELETE) and **endpoints** (URLs).

Key Features:

- Uses HTTP verbs
- Resource-based (like `/users` , `/products`)
- Stateless (no client data stored on server)
- Supports JSON, XML (usually JSON)
- Easy to use & scalable

Example:

Operation	Endpoint	Method
Get all users	<code>/api/users</code>	GET

Operation	Endpoint	Method
Create user	<code>/api/users</code>	POST
Get user	<code>/api/users/:id</code>	GET
Update user	<code>/api/users/:id</code>	PUT
Delete user	<code>/api/users/:id</code>	DELETE

✓ You're using this in your project (EXPENSE TRACKER MERN).

2. GraphQL API

✓ What it is:

A **query language** for your API. Instead of hitting many endpoints, you send a **single query** to get exactly what you need.

Key Features:

- Single endpoint (usually `/graphql`)
- Request only the data you need
- Reduces over-fetching and under-fetching
- Requires a schema
- Great for frontend flexibility

Example:

graphql

```
query {  
  user(id: "123") {  
    name  
    email  
    posts {  
      title  
    }  
  }  
}
```



```
}  
}
```

➡ One query gets `user`, `email`, and their `posts` in one go!

📖 3. gRPC (Google Remote Procedure Call)

✅ What it is:

A high-performance open-source RPC framework by Google. Mostly used for **microservices** and **backend-to-backend** communication.

🔧 Key Features:

- Uses **Protocol Buffers** (binary format)
- Super fast and lightweight
- Strongly typed
- Supports multiple languages (Go, Java, Python, etc.)
- Not used in browsers directly

🧠 Example:

Instead of sending HTTP requests, you define services like:

```
proto  
  
service UserService {  
  rpc GetUser (UserRequest) returns (UserResponse);  
}
```

📖 4. SOAP (Simple Object Access Protocol)

✅ What it is:

An **older XML-based** protocol for accessing web services.

Key Features:

- Uses XML for messaging
 - More secure and standardized
 - Used in **enterprise**, banks, or government systems
 - Heavy and strict in structure
 - Rarely used in modern web apps
-

5. WebSockets

What it is:

A real-time, two-way communication protocol over a single TCP connection.

Key Features:

- Full-duplex communication (client & server can send anytime)
 - Great for chat apps, games, live dashboards
 - Not RESTful
 - Persistent connection
-

Summary Table

API Type	Endpoint Style	Format	Use Case	Pros
REST	Multiple URLs	JSON	Most web apps	Easy, widely supported
GraphQL	Single URL	JSON	Dynamic frontend, mobile apps	Flexible, efficient
gRPC	Binary (proto)	Protobuf	Microservices, high-speed apps	Fast, strongly typed
SOAP	WSDL (XML)	XML	Enterprise, legacy systems	Secure, standard

API Type	Endpoint Style	Format	Use Case	Pros
WebSocket	Persistent conn	Custom msgs	Real-time apps (chat, notifications)	Real-time, bidirectional

✓ Which One Should You Use?

Project Type	Best API Type
Standard Web App (CRUD)	✓ REST
Highly Dynamic Data (React)	✓ GraphQL
Real-Time (Chat/Game)	✓ WebSockets
Microservices Backend	✓ gRPC
Government/Legacy Systems	✓ SOAP

Want a YAML or visual diagram flow of when to use which API or how they compare in structure? I can make that for you next!

part of Axios' features and Axios' advanced features

Great question, Prasun! Let's break this down into two parts:

📖 Part of Axios' Features (Core Features)

These are the **basic and essential features** of Axios you use in almost every project:

Feature	Description
1. Promise-based	Axios returns a promise which simplifies async code with <code>.then()</code> / <code>async-await</code> .
2. Request Methods	Supports <code>GET</code> , <code>POST</code> , <code>PUT</code> , <code>PATCH</code> , <code>DELETE</code> , etc.
3. Automatic JSON Data	Automatically transforms request & response to/from JSON.
4. Request Configuration	Allows <code>baseUrl</code> , <code>headers</code> , <code>params</code> , <code>timeout</code> , etc.
5. Interceptors	Lets you intercept requests or responses globally (e.g., to attach tokens).
6. Error Handling	Clean error handling with <code>try/catch</code> or <code>.catch()</code> .
7. Cancellation	Cancel requests using <code>CancelToken</code> .
8. CSRF Protection	Can automatically handle CSRF tokens when set up properly.

Axios' Advanced Features

These are **powerful enhancements** often used in large-scale applications or for fine-grained control:

Feature	Description
1. Axios Instance	Create custom instances of Axios with predefined configs (like <code>baseUrl</code> , <code>headers</code>).
2. Interceptors (Advanced Use)	Use multiple request/response interceptors for auth, logging, error tracking, etc.
3. Request & Response Transformation	Modify data before sending (request) or after receiving (response) using <code>transformRequest</code> , <code>transformResponse</code> .
4. Global Defaults	Set default configs for all Axios requests globally (like headers, timeout, etc.).
5. Multiple Requests (Concurrency)	Use <code>axios.all()</code> or <code>Promise.all()</code> to send multiple requests in parallel.
6. Custom Headers	Send custom tokens, session IDs, or any user-defined headers.

Feature	Description
7. Progress Tracking	Track upload/download progress using <code>onUploadProgress</code> and <code>onDownloadProgress</code> .
8. Response Schema Control	Check <code>response.status</code> , <code>response.data</code> , <code>response.headers</code> , etc., to control behavior based on backend.
9. Retry Mechanism <i>(manually or with plugin)</i>	Retry failed requests with a delay or backoff.

✓ Example Use of Advanced Axios Features

js

```
const axiosInstance = axios.create({
  baseURL: "https://api.example.com",
  timeout: 5000,
  headers: {
    "Content-Type": "application/json",
  },
  transformRequest: [(data) => {
    // Transform before request
    return JSON.stringify(data);
  }],
  transformResponse: [(data) => {
    // Transform after response
    return JSON.parse(data);
  }]
});

// Add request interceptor
axiosInstance.interceptors.request.use((config) => {
  const token = localStorage.getItem("token");
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});
```

If you want, I can also make a **YAML flow or visual chart** showing how core and advanced Axios features flow in a real project. Just say the word!

part of fetch' features and fetch' advanced features

Certainly, Prasun! Let's break down **Fetch API** into its **core** and **advanced** features.

Core Features of Fetch API

These are the **basic features** that Fetch provides for sending requests and handling responses:

Feature	Description
1. Native Browser Support	<code>fetch</code> is a native JavaScript API supported in modern browsers, making it easy to use without external libraries.
2. Promise-based	Fetch is promise-based, meaning you can use <code>.then()</code> and <code>.catch()</code> or <code>async/await</code> to handle responses.
3. Supports GET and POST Requests	Common HTTP methods like GET and POST are supported out of the box.
4. Handles Response Data	Automatically parses response data (can be JSON, text, etc.).
5. Request Configuration	You can pass various options (method, headers, body) to customize the request.
6. JSON Support	Fetch allows you to directly work with JSON using <code>response.json()</code> to parse JSON data from the server.
7. Streams	Fetch supports Streams , meaning you can process large files or data chunks while they're being downloaded.
8. Simple Error Handling	Fetch will not reject the promise on HTTP errors (4xx, 5xx). You need to check for errors manually (e.g., using <code>response.ok</code>).

Advanced Features of Fetch API

These are the **advanced capabilities** that Fetch supports, which allow for more control over the request/response process and handling of specific use cases.

Feature	Description
1. Custom Headers	You can pass custom headers (e.g., for authentication) via the <code>headers</code> property in the <code>fetch</code> options.
2. Request Method Customization	You can customize the HTTP method (GET, POST, PUT, DELETE, PATCH) in the request options.
3. Body Support for POST/PUT	You can send data in the body of the request (e.g., sending JSON or form data).
4. Handling Non-200 Responses	Fetch does not reject on HTTP errors. You must manually check <code>response.ok</code> to handle errors.
5. Abort Controller	Allows you to cancel a request using an <code>AbortController</code> .
6. CORS (Cross-Origin Resource Sharing)	Fetch supports CORS, which is necessary for handling cross-origin requests and security.
7. Response Types (Text, Blob, JSON, etc.)	Fetch allows you to choose how to handle the response body, such as <code>text()</code> , <code>json()</code> , <code>blob()</code> , etc.
8. Cache Control	You can control caching behaviors using the <code>cache</code> option (e.g., <code>no-store</code> , <code>reload</code> , etc.).
9. Credentials	The <code>credentials</code> option allows you to send cookies and authentication data in cross-origin requests.
10. Retry Logic	Fetch doesn't support retrying failed requests out of the box, but you can implement it manually (for example, by using <code>setTimeout</code> or <code>Exponential Backoff</code>).

Example Use of Advanced Fetch Features

Example: Handling Request and Response with Fetch

js

```
const url = 'https://api.example.com/data';
const requestOptions = {
  method: 'GET', // Or 'POST', 'PUT', etc.
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token_here',
  },
  credentials: 'same-origin', // For sending cookies with requests
  cache: 'no-cache', // Disable caching
};

// Fetch request
fetch(url, requestOptions)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Parse JSON response
  })
  .then(data => console.log(data))
  .catch(error => console.error('There was an error!', error));
```

Advanced Example: Using `AbortController` to Cancel Fetch Request

js

```
const controller = new AbortController();
const signal = controller.signal;

fetch('https://api.example.com/data', { signal })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => {
    if (err.name === 'AbortError') {
      console.log('Fetch request was aborted');
    } else {
      console.error('Fetch error:', err);
    }
  });
```



```
// Abort the request after 5 seconds
setTimeout(() => controller.abort(), 5000);
```

✓ Summary of Features Comparison

Feature	Fetch API	Axios
Promise-based	✓	✓
Support for HTTP Methods	✓ (GET, POST, etc.)	✓ (GET, POST, etc.)
Request Headers	✓	✓
Request Body	✓ (with <code>POST</code> , <code>PUT</code> , etc.)	✓ (with <code>POST</code> , <code>PUT</code> , etc.)
Error Handling	Manual check of <code>response.ok</code>	Automatically throws on error
Automatic JSON Parsing	✓ (<code>response.json()</code>)	✓ (automatic JSON parsing)
CORS Support	✓	✓
Cancellation Support	✓ (with <code>AbortController</code>)	✓ (with <code>CancelToken</code>)
Request & Response Interceptors	✗	✓
Progress Tracking	✗	✓

🧠 Which One Should You Choose?

- **Fetch API:** Great for simple requests, native in modern browsers, and easy to use.
- **Axios:** A more feature-rich solution (supports request/response interceptors, cancellation tokens, and global configuration).

I hope this clarifies the core and advanced features of **Fetch API**. Would you like a flowchart or detailed breakdown of how these features are used in a real project? Let me know!

```
import axios from "axios";
import { BASE_URL } from "../apiPaths";

// Axios API Instance
const axiosInstance = axios.create({
  baseURL: BASE_URL,
  timeout: 10000,
  headers: {
    "Content-Type": "application/json",
    Accept: "application/json",
  },
});

// Request Interceptor same as Postman we have to do manually
// login --> then give Token in the Variables --> then work.
axiosInstance.interceptors.request.use(
  (config) => {
    const accessToken = localStorage.getItem("token");
    if (accessToken) {
      config.headers.Authorization = `Bearer ${accessToken}`;
    }

    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response Interceptor
axiosInstance.interceptors.response.use(
  (response) => {
    return response;
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

```
// Handle common errors globally
if (error.response) {
  if (error.response.status === 401) {
    // Redirect to Login page
    window.location.href = "/login";
  } else if (error.response.status === 500) {
    console.error("Server error. Please try again later.");
  }
  } else if (error.code === "ECONNABORTED") {
    console.error("Request timeout. Please try again.");
  }
  return Promise.reject(error);
}
);

export default axiosInstance;
```

How this things is done using the fetch()

To achieve similar functionality in `fetch()` as you have with `axios` (such as request interceptors, adding authentication tokens, and handling errors globally), you need to manually handle things like headers and response handling, as `fetch()` doesn't provide built-in interceptors.

Below is an equivalent implementation of your **Axios** code using the `fetch()` API:

Steps to Implement:

1. Set Up Request with Token Authentication
2. Handle Global Errors
3. Handle Response Parsing

Equivalent Code Using `fetch()`

javascript

```
// Utility to get the token from localStorage and set it in headers
const getAuthToken = () => {
  const token = localStorage.getItem("token");
  return token ? `Bearer ${token}` : null;
};
```

```

// Request function using fetch
const request = async (url, options = {}) => {
  const token = getAuthToken();

  // Default headers
  const headers = {
    "Content-Type": "application/json",
    Accept: "application/json",
    ...options.headers, // Allow custom headers passed to function
  };

  if (token) {
    headers.Authorization = token;
  }

  // Setting up the request options
  const config = {
    method: options.method || "GET", // Default method is GET
    headers: headers,
    body: options.body ? JSON.stringify(options.body) : null, // Stringify the body
    for POST/PUT requests
  };

  try {
    // Send the request using fetch
    const response = await fetch(url, config);

    // Check if response is ok (status 200-299)
    if (!response.ok) {
      if (response.status === 401) {
        // Handle Unauthorized (401) - Redirect to Login
        window.location.href = "/login";
      } else if (response.status === 500) {
        console.error("Server error. Please try again later.");
      }
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    // If response is okay, parse the response body based on content type
    const responseData = await response.json();
    return responseData;
  }

```

```

} catch (error) {
  // Handle other errors (like network issues or timeouts)
  if (error.name === "AbortError") {
    console.error("Request was aborted");
  } else {
    console.error("Fetch error:", error);
  }
  throw error; // Re-throw the error to be handled in the calling function
}
};

// Example of using the request function
const fetchData = async () => {
  try {
    const data = await request('/api/v1/dashboard', { method: 'GET' });
    console.log("Data:", data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
};

fetchData();

```

Key Differences and Features:

1. Setting Authorization Token:

- In `axios`, the request interceptor automatically attaches the token to the headers. In `fetch()`, we manually retrieve the token from `localStorage` and set it in the `Authorization` header.

2. Global Error Handling:

- `axios` provides automatic error handling for HTTP status codes (e.g., 401, 500). With `fetch()`, we have to manually check `response.ok` to ensure the request was successful. If not, we handle errors accordingly.

3. Request Body:

- `fetch()` does not automatically transform the body into JSON like `axios`. You must manually convert the request body (if present) using `JSON.stringify()`.

4. Response Parsing:

- In `axios`, the response body is automatically parsed as JSON. In `fetch()`, you need to explicitly parse the response using `.json()` or `.text()` based on the content type.

Features Added in This Implementation:

1. Authorization Token:

The token is attached to the headers for any request.

2. Custom Method Support:

The method (GET, POST, PUT, etc.) is customizable through the `options` parameter.

3. Error Handling:

- If the status is 401 (Unauthorized), the user is redirected to the login page.
- If the status is 500 (Server Error), a message is logged in the console.
- All errors are caught in the `catch` block and logged.

4. Global Error Handling for Request Timeout:

You could use an `AbortController` to manually handle request timeouts if needed, but `fetch()` does not natively support this without an `AbortController`.

Explanation of Code:

- `getAuthToken` function: Retrieves the token from `localStorage` and returns it in the format `Bearer <token>`. If no token is present, it returns `null`.
- `request` function: This is the main function that abstracts the logic for making a `fetch()` call. It allows passing custom request options (e.g., method, body, headers).
- **Error handling:** Errors like 401 or 500 are handled manually, and generic errors (e.g., network errors) are caught using a `try-catch` block.

Comparison with Axios:

Feature	Axios	Fetch
Automatic JSON Parsing	✅ Automatically parses JSON	❌ Must manually parse response (<code>response.json()</code>)
Request Interceptors	✅ Built-in support	❌ Must manually handle headers and token injection
Response Interceptors	✅ Built-in support	❌ Must handle errors manually
Global Error Handling	✅ Built-in support	❌ Must manually handle error status codes
Abort Request	✅ <code>CancelToken</code> support	✅ <code>AbortController</code> (manually implemented)
Customizable Timeout	✅ Built-in support	✅ Must manually implement using <code>AbortController</code>
Browser Support	✅ Cross-browser support	✅ Cross-browser support

This is how you can manually implement similar functionality to `axios` using `fetch()`. Let me know if you need further details!

Its really long and axios is not that much long

Yes, you're absolutely right! `Axios` makes things simpler and more compact, while with `fetch()`, you need to manually handle a lot of things like setting headers, parsing the response, and handling errors. This leads to more verbose code.

Let's compare the simplified version of both:

Using Axios (Simplified)

javascript

```
import axios from "axios";
import { BASE_URL } from "../apiPaths";

// Axios API Instance
const axiosInstance = axios.create({
  baseURL: BASE_URL,
  timeout: 10000,
  headers: {
```

```

    "Content-Type": "application/json",
    Accept: "application/json",
  },
});

// Request Interceptor (Token handling)
axiosInstance.interceptors.request.use((config) => {
  const accessToken = localStorage.getItem("token");
  if (accessToken) config.headers.Authorization = `Bearer ${accessToken}`;
  return config;
});

// Response Interceptor (Error handling)
axiosInstance.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) window.location.href = "/login";
    return Promise.reject(error);
  }
);

export default axiosInstance;

```

Using Fetch (Simplified)

javascript

```

// Utility to get token from localStorage
const getAuthToken = () => localStorage.getItem("token");

// Fetch Request Function
const fetchData = async (url, method = "GET", body = null) => {
  const token = getAuthToken();
  const headers = {
    "Content-Type": "application/json",
    Accept: "application/json",
    Authorization: token ? `Bearer ${token}` : "",
  };

  const response = await fetch(url, {
    method,
    headers,
  });

```



```
    body: body ? JSON.stringify(body) : null,
  });

  if (!response.ok) {
    if (response.status === 401) window.location.href = "/login";
    throw new Error("Request failed");
  }

  return response.json(); // parse the JSON response
};

export default fetchData;
```

Comparison:

- **Length:** Axios is **shorter and more readable** since it abstracts away a lot of things (like handling headers and errors).
- **Flexibility:** `fetch()` gives you more control, but you need to manage things manually (headers, errors, etc.).
- **Error Handling:** In Axios, it's automatic with interceptors, while in `fetch()`, you need to manually check the response status and handle errors.

In conclusion, **Axios** is more **developer-friendly** and **convenient** for common tasks (like authentication headers, error handling, and automatic JSON parsing). On the other hand, `fetch()` is more **lightweight** and **standardized**, but requires more effort to handle things like interceptors and error management.

So, you're right in feeling that Axios is **more concise and easy to use**, especially in a full-fledged project where you have more complex needs (like authentication, token handling, etc.).

Let me know if you'd like to dive into any other comparison points!