React Question - Week 5 (Aug11- Aug-17)

Compiled by	Ⓚ Kumar Nayan

❓ Q #1 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture, 🔍 Real-World Scenario

🔥 EXTREMELY IMPORTANT

🧠 Question:

What are the key differences between `fetch` and `Axios` for making HTTP requests in a React application?

✅ Answer:

The primary difference is that `fetch` is a native browser API, while Axios is a third-party library that needs to be installed. Axios offers more features out-of-the-box, making it more convenient for complex applications.

Key differences mentioned in your notes include:

1. **JSON Data Handling**: `fetch` requires a two-step process to handle JSON (first getting the response, then calling `.json()`), whereas Axios automatically parses JSON responses.
2. **Headers**: Axios automatically sets headers like `Content-Type` for POST requests, while with `fetch` you must configure them manually.

3. **Error Handling**: `fetch` does not consider HTTP error statuses (like 404 or 500) as promise rejections, so you have to check `res.ok` manually. Axios rejects the promise on these error statuses, making error handling more straightforward.
4. **Browser Support**: Axios has built-in support for older browsers, whereas `fetch` may require a polyfill.

```javascript
// Fetch GET request (requires .json())
fetch("https://jsonplaceholder.typicode.com/users")
  .then(res => res.json())
  .then(data => console.log(data));

// Axios GET request (data is already parsed in res.data)
axios.get("https://jsonplaceholder.typicode.com/users")
  .then(res => console.log(res.data));
```

❓ Q #2 (🟡 Intermediate)

Tags: 🧩 General Interview, 🔧 Hooks, 💥 Pitfall

🔥 EXTREMELY IMPORTANT

🧠 Question:

How do you cancel an in-flight `fetch` request within a React component, and why is this important in a `useEffect` hook?

✅ Answer:

You can cancel a `fetch` request using the `AbortController` API. You create an instance of `AbortController`, pass its `signal` property to the `fetch` options, and then call `controller.abort()` when you want to cancel it.

This is crucial in a `useEffect` hook to prevent memory leaks and race conditions. If a component unmounts while a `fetch` request is still pending, the request will eventually complete and try to update the state of an unmounted component, causing a React warning and potential bugs. The cleanup function of `useEffect` is the perfect place to call `controller.abort()`.

```jsx
useEffect(() => {
  const controller = new AbortController();
  const signal = controller.signal;

  fetch('https://jsonplaceholder.typicode.com/users', { signal })
    .then(res => res.json())
    .then(data => setData(data))
    .catch(err => {
      if (err.name === 'AbortError') {
        console.log('Fetch aborted');
```

```
    }
  });

  // Cleanup function: abort the request if the component unmounts
  return () => controller.abort();
}, []);
```

❓ Q #3 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🔥 EXTREMELY IMPORTANT

🧠 Question:

From a frontend developer's perspective, what are the fundamental differences between REST and GraphQL?

✅ Answer:

The fundamental difference lies in how data is requested and received.

REST is an architectural style based on predefined endpoints. Each endpoint (`/users`, `/posts/1`) represents a specific resource and typically returns a fixed data structure. To get related data, like a user and all their posts, you often need to make multiple requests (e.g., one to `/users/1` and another to `/users/1/posts`).

GraphQL is a query language for APIs. It uses a single endpoint (e.g., `/graphql`) where the client sends a query specifying the exact data and structure it needs. This allows you to fetch a user and their posts in a single request, solving the problems of over-fetching (getting more data than needed) and under-fetching (not getting enough data, requiring more calls).

```javascript
// GraphQL query to get a user and their posts in one request
const query = `
  query {
    user(id: 1) {
      id
      name
      posts {
        title
      }
    }
  }
`;
```

❓ Q #4 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🔥 EXTREMELY IMPORTANT

🧠 Question:

Where should you store authentication tokens (like JWTs) on the frontend? Discuss the security trade-offs between `localStorage` and `HttpOnly` cookies.

✅ Answer:

Storing tokens securely is critical. The two main options have significant security trade-offs:

1. `localStorage`:
    - **Pros**: Easy to access with JavaScript. Persists across browser sessions.
    - **Cons**: Highly vulnerable to Cross-Site Scripting (XSS) attacks. If a malicious script is injected into your site, it can read the token from `localStorage` and impersonate the user. This is a major security risk.
2. `HttpOnly` **Cookies**:
    - **Pros**: Far more secure against XSS. The `HttpOnly` flag prevents any client-side JavaScript from accessing the cookie, effectively blocking token theft via XSS.
    - **Cons**: Can be vulnerable to Cross-Site Request Forgery (CSRF) attacks, where a malicious site tricks the user's browser into making a request to your app, and the browser automatically includes the cookie. This risk can be mitigated by setting the `SameSite=Strict` or `SameSite=Lax` flag on the cookie and using anti-CSRF tokens.

For security, `HttpOnly` **cookies are the recommended approach** for storing authentication tokens, provided CSRF protections are in place.

❓ Q #5 (🟡 Intermediate)

Tags: 🧩 General Interview, 🌐 Routing, 🧱 Components

🧠 Question:

How would you implement a `PrivateRoute` component in a React application to protect routes from unauthenticated users?

✅ Answer:

A `PrivateRoute` component acts as a wrapper around a route. It checks for the existence or validity of an authentication token. If the user is authenticated, it

renders the intended component. If not, it redirects the user to a public page, like the login screen.

This is typically implemented using `react-router-dom`. The component checks a condition (e.g., `!!localStorage.getItem('authToken')`) and conditionally renders either the protected component or a `<Redirect>` (or `<Navigate>` in v6).

```jsx
// Example using react-router-dom v5
import { Route, Redirect } from 'react-router-dom';

const PrivateRoute = ({ component: Component, ...rest }) => {
  // This check should ideally be more robust (e.g., validating the token)
  const isAuthenticated = !!localStorage.getItem('authToken');

  return (
    <Route
      {...rest}
      render={props =>
        isAuthenticated ? (
          <Component {...props} />
        ) : (
          <Redirect to="/login" />
        )
      }
    />
  );
};
```

❓ Q #6 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is a JSON Web Token (JWT) and what are its three main parts?

✅ Answer:

A JSON Web Token (JWT) is a compact, URL-safe standard for securely transmitting information between a client and a server as a JSON object. It is commonly used for stateless authentication because it contains all the necessary user information and is digitally signed to verify its integrity.

A JWT consists of three parts separated by dots (`.`):

1. **Header**: Contains metadata about the token, such as the signing algorithm used (e.g., `HS256`) and the token type (`JWT`).
2. **Payload**: Contains the claims or data about the user (e.g., `userId`, `role`, `exp` - expiration time). This is the information the server will use.

3. **Signature**: A cryptographic signature created using the encoded header, the encoded payload, and a secret key known only to the server. This signature ensures that the token has not been tampered with.

❓ Q #7 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

Explain the difference between stateful and stateless authentication.

✅ Answer:

The key difference is where the user's session data is stored.

Stateful Authentication (e.g., traditional sessions) requires the server to store session information for every logged-in user, usually in memory or a database. The client receives a session ID (often in a cookie) and sends it with each request. The server then looks up this ID in its storage to identify the user. This approach can be difficult to scale across multiple servers, as they would need to share session storage.

Stateless Authentication (e.g., JWT) does not require the server to store any session information. After login, the server generates a token (like a JWT) that contains all the necessary user data and sends it to the client. The client includes this token in every subsequent request. The server can validate the token's signature to verify the user's identity without needing to look up anything in storage. This makes the system more scalable and simpler to manage.

❓ Q #8 (🟢 Basic)

Tags: 🧩 General Interview, 🌐 Routing, 🏗️ Architecture

🔥 EXTREMELY IMPORTANT

🧠 Question:

What is CORS, and why is it important for a frontend developer to understand?

✅ Answer:

CORS stands for **Cross-Origin Resource Sharing**. It is a security mechanism built into web browsers that restricts a web page from making requests to a

different domain (origin) than the one that served the page. By default, browsers block these cross-origin requests.

For a frontend developer, this is critical because modern applications often have a frontend (e.g., a React app on `localhost:3000`) that needs to fetch data from a backend API running on a different origin (e.g., `localhost:3001` or a public API). Without proper CORS configuration on the **server**, the browser will block the API requests, and your application will fail to load data. You need to ensure the backend server sends the correct CORS headers (like `Access-Control-Allow-Origin`) to permit requests from your frontend's domain.

❓ Q #9 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 📦 State Management, 🌐 Routing

🧠 Question:

What is the purpose of Apollo Client, and how does its `useQuery` hook simplify data fetching in a GraphQL-powered React app?

✅ Answer:

Apollo Client is a comprehensive state management library for JavaScript that enables you to manage both remote data with GraphQL and local data. It provides caching, UI updates, and error handling out-of-the-box.

The `useQuery` **hook** is a core feature that drastically simplifies fetching data. It handles the entire lifecycle of a request:

1. It sends the GraphQL query to the server.
2. It tracks the request's state, returning `loading`, `error`, and `data` properties that you can use to conditionally render your UI.
3. It automatically caches the response, so subsequent requests for the same data can be resolved from the cache instantly, making the UI feel faster.

   This declarative approach eliminates the boilerplate code of manual state handling (`useState`, `useEffect`) that you would write with `fetch`.

```
jsximport { useQuery, gql } from "@apollo/client";

const GET_REPOS = gql`
  query GetRepos { viewer { repositories(last: 10) { nodes { name } } } }
`;

function App() {
    // useQuery handles all the state for you
```

```
  const { loading, error, data } = useQuery(GET_REPOS);

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
    <ul>
      {data.viewer.repositories.nodes.map(repo => <li key={repo.name}>{repo.name}</li>)}
    </ul>
  );
}
```

❓ Q #10 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

In what scenarios might you prefer using a REST API over GraphQL?

✅ Answer:

While GraphQL is powerful, REST is often a better choice in certain scenarios:

1. **Simple APIs**: For applications with straightforward data needs and few entities, REST is simpler to implement and maintain. The overhead of setting up a GraphQL schema and resolvers might be unnecessary.
2. **HTTP Caching**: REST APIs can leverage browser and CDN caching very effectively using standard HTTP methods (GET) and headers. Caching in GraphQL is more complex and typically handled at the client level (e.g., Apollo Client's cache).
3. **File Uploads**: While possible with GraphQL, handling file uploads is often more direct and simpler using a standard RESTful `multipart/form-data` request.
4. **Performance with Complex Queries**: A poorly designed or malicious GraphQL query can be very resource-intensive on the server, potentially causing performance issues. REST endpoints have a more predictable and controlled server load.

❓ Q #11 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

What is the purpose of a refresh token in an authentication system?

✅ Answer:

A refresh token is used to obtain a new access token without requiring the user to log in again. This is a crucial security practice.

Access tokens (like JWTs) are typically short-lived (e.g., 5-15 minutes). This limits the damage if one is stolen, as it will expire quickly. However, you don't want to force the user to re-authenticate every 15 minutes.

This is where the **refresh token** comes in. It is a long-lived token that is stored securely (ideally in an `HttpOnly` cookie). When the access token expires, the client can send the refresh token to a special endpoint (e.g., `/refresh_token`). The server validates the refresh token and, if valid, issues a new short-lived access token. This flow provides a seamless user experience while maintaining a high level of security.

❓ Q #12 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is the difference between Node.js and Express.js?

✅ Answer:

Node.js is a JavaScript runtime environment. It allows you to run JavaScript code outside of a web browser, on a server. It provides low-level features for handling tasks like file system access, networking, and event handling. You can build a web server with just Node.js, but it requires writing a lot of boilerplate code.

Express.js is a web application framework that runs on top of Node.js. It is not a separate language or runtime; it's a library that simplifies the process of building web servers and APIs with Node.js. It provides helpful abstractions for common tasks like routing (handling different URLs), middleware (processing requests), and managing request/response cycles, resulting in much cleaner and more organized code.

❓ Q #13 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

Explain what a CSRF attack is and how `SameSite` cookie attributes can help mitigate it.

✅ Answer:

A **Cross-Site Request Forgery (CSRF)** attack tricks an authenticated user's browser into sending a malicious request to a web application they are logged into. For example, a user visits a malicious site `evil.com`, which contains an invisible form that automatically submits a request to `your-bank.com/transfer?amount=1000`. If the user is logged into their bank and the bank uses cookies for session management, the browser will automatically include the authentication cookie with the request, and the bank will process the transfer.

The `SameSite` cookie attribute is a powerful defense against CSRF. It tells the browser whether to send a cookie with cross-site requests.

- `SameSite=Strict`: The cookie will only be sent if the request originates from the same site. This provides the strongest protection but can break legitimate cross-site navigation (e.g., clicking a link to your site from an email).
- `SameSite=Lax`: A good balance. The cookie is sent with top-level navigations (e.g., clicking a link) but is withheld on cross-origin subrequests like those made by forms, iframes, or AJAX. This is the default in modern browsers.

❓ Q #14 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

From a frontend perspective, what is the role of middleware in an Express.js application?

✅ Answer:

Middleware functions are functions that have access to the request (`req`), response (`res`), and the `next` function in the application's request-response cycle. They can execute code, make changes to the request and response objects, end the request-response cycle, or call the next middleware in the stack.

From a frontend developer's perspective, middleware on the backend is important for handling cross-cutting concerns that affect your API calls, such as:

1. **Authentication**: A middleware function can check for a valid JWT in the `Authorization` header and either allow the request to proceed to the route handler or block it if the token is missing or invalid.
2. **CORS**: `cors()` is a middleware that adds the necessary HTTP headers to allow your frontend application to make cross-origin requests.
3. **JSON Parsing**: `express.json()` is middleware that parses incoming JSON request bodies, making the data available on `req.body` for POST or PUT requests sent from your React app.

```javascript
// Example of an authentication middleware in Express
function authenticate(req, res, next) {
  const token = req.headers["authorization"]?.split(' ')[1];
  if (!token) {
    return res.status(401).json({ message: "unauthorized" });
  }
  try {
    const validated = jwt.verify(token, process.env.token);
    req.user = validated;
    next();  // Pass control to the next function
  } catch(e) {
    return res.status(401).json({ message: "unauthorized" });
  }
}

// Usage: The `authenticate` middleware runs before the route handler
router.get("/protected", authenticate, (req, res) => {
  res.json({ message: "successful" });
});
```

❓ Q #15 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 🧪 Testing

🧠 Question:

In a TypeScript project, how can you ensure your frontend types accurately reflect the API responses from the backend?

✅ Answer:

Keeping frontend types in sync with the backend API is crucial for type safety. Your notes mention three effective methods:

1. **OpenAPI/Swagger Specification**: If the backend provides an OpenAPI (formerly Swagger) spec file (e.g., `swagger.json`), you can use tools like `openapi-typescript` to automatically generate TypeScript type definitions from it. This is a robust, automated way to maintain synchronization.

2. **Manual Type Definition and Inference**: You can manually define types for API responses. To improve on this, you can use TypeScript's inference capabilities. For example, using `Awaited<typeof axiosCall>` lets you infer the full response type directly from an actual Axios call, which is useful when you have a working endpoint.
3. **Shared Types**: In a monorepo setup where both frontend and backend are written in TypeScript, you can define the types in a shared package and import them into both projects, ensuring they are always in sync.

```typescript
// Example of inferring from an Axios response
const res = await axios.get<Product[]>('/api/products');

// 'ProductsResponse' will be inferred as Product[]
type ProductsResponse = typeof res.data;

// 'LoginResponse' will be inferred as AxiosResponse<LoginToken>
type LoginResponse = Awaited<typeof loginRes>;
```

❓ Q #16 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is an ORM like Prisma, and why would a backend team choose to use it instead of writing raw SQL queries?

✅ Answer:

An ORM, or **Object-Relational Mapper**, is a tool that creates a "bridge" between an object-oriented programming language (like JavaScript/TypeScript) and a relational database (like PostgreSQL). Prisma allows developers to interact with the database by writing familiar JavaScript/TypeScript code instead of raw SQL.

A backend team would choose Prisma for several key reasons:

1. **Type Safety**: This is a major benefit in TypeScript projects. Prisma generates a client based on your database schema, so you get full type checking and autocompletion for your database queries. This catches errors at compile time, not runtime (e.g., typos in column names).
2. **Developer Productivity**: It's often faster to write `prisma.user.create(...)` than a full `INSERT INTO "users" (...)` statement. It abstracts away the complexities of SQL, allowing developers to focus on application logic.

3. **Improved Readability and Maintainability**: ORM code is often more readable and easier to understand for developers who may not be SQL experts.
4. **Migration Management**: Prisma provides a powerful system for managing database schema changes (migrations), making it easy to keep the database structure in sync with the application code.

❓ Q #17 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is the purpose of the `schema.prisma` file when using the Prisma ORM?

✅ Answer:

The `schema.prisma` file is the central "source of truth" for your database schema and Prisma setup. It serves as a blueprint for your database.

It has three main sections:

1. **Generator**: Specifies what client should be generated. For a Node.js project, this is `prisma-client-js`.
2. **Datasource**: Defines your database connection, including the type of database (e.g., `postgresql`) and the connection URL, which is usually stored in an environment variable.
3. **Models**: This is where you define your database tables as models. Each model maps to a table, and its fields map to columns. You define field types, primary keys, relationships between tables, and other constraints here. Prisma uses these models to generate the type-safe client.

❓ Q #18 (🟡 Intermediate)

Tags: 🧩 General Interview, 🔧 Hooks, 🏗️ Architecture

🧠 Question:

You created a `useFetch` custom hook in your notes. What are the primary benefits of abstracting data-fetching logic into a custom hook like this?

✅ Answer:

Abstracting data-fetching logic into a custom hook like `useFetch` offers several significant benefits:

1. **Reusability**: You can reuse the same data-fetching logic across multiple components without duplicating code.
2. **Separation of Concerns**: It separates the complex logic of fetching, state management (`loading`, `error`, `data`), and cleanup from the component's rendering logic. This makes components cleaner and easier to read.
3. **Centralized Logic**: Any changes or improvements to the data-fetching logic (e.g., adding caching, improving error handling) only need to be made in one place—the custom hook.
4. **Simplified Component Code**: Components that use the hook become much simpler. They just need to call `const { data, loading, error } = useFetch(url);` and can immediately use these states to render the UI, without worrying about the implementation details.

❓ Q #19 (🟢 Basic)

Tags: 🧩 General Interview, 📦 State Management, 🌐 Routing

🧠 Question:

What is the role of the `<ApolloProvider>` component when setting up Apollo Client in a React application?

✅ Answer:

The `<ApolloProvider>` component is a wrapper component from Apollo Client that uses React's Context API to make an instance of the Apollo Client available to any component in the component tree below it.

You typically wrap your entire `<App />` component with `<ApolloProvider>`. This ensures that any component in your application, no matter how deeply nested, can access the client to execute GraphQL queries or mutations using hooks like `useQuery` or `useMutation`. It's the central connection point between your React application and your GraphQL API.

```jsx
// In your main entry file, like index.js or main.jsx
const client = new ApolloClient({ /* ...configuration... */ });

createRoot(document.getElementById("root")).render(
  <ApolloProvider client={client}>
    <App />
```

```
</ApolloProvider>
);
```

❓ Q #20 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is the difference between "database migration" and "database seeding"?

✅ Answer:

Database Migration refers to the process of managing and applying changes to your database schema over time. When you change your application code and need to alter the database structure (e.g., add a new table, add a column to an existing table, or change a data type), you create a migration file. Running the migration applies these changes to the database. Tools like Prisma automate the generation and application of these migration files.

Database Seeding is the process of populating a database with initial or dummy data. This is extremely useful for development and testing, as it allows you to start with a consistent set of data (e.g., sample users, products) every time you set up your environment. You write a seed script that creates this data, and then run it to populate the tables.

❓ Q #21 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

How does a frontend application typically make a request to a GraphQL API, and how does this differ from REST?

✅ Answer:

The request pattern is fundamentally different.

With **REST**, you make requests to various endpoints using different HTTP methods that correspond to CRUD actions (e.g., `GET /users`, `POST /users`, `DELETE /users/1`).

With **GraphQL**, you almost always send a `POST` request to a **single endpoint** (e.g., `/graphql`). The body of this POST request contains a JSON object with a

`query` property. The value of this `query` property is a string containing the GraphQL operation (query or mutation) you want to execute. The server parses this string and returns the data that matches the query's shape. This means the client, not the server's endpoint structure, dictates what data is returned.

```javascript
// Example of a GraphQL request from the frontend using fetch
async function getAllUsers() {
  const query = `
    query {
      users {
        id
        name
      }
    }
  `;

  const res = await fetch("http://localhost:4000/graphql", {
    method: "POST", // Always POST
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ query }) // The query is in the body
  });

  const { data } = await res.json();
  console.log(data.users);
}
```

❓ Q #22 (🟢 Basic)

Tags: 🧩 General Interview, 💥 Pitfall

🧠 Question:

When making a POST request using the native `fetch` API, why is it important to include the `Content-Type: 'application/json'` header?

✅ Answer:

It's important to include the `Content-Type: 'application/json'` header to inform the server about the type of data you are sending in the request body.

When you send a JavaScript object as the body of a POST request, you first need to serialize it into a JSON string using `JSON.stringify()`. This header tells the server's body-parsing middleware (like `express.json()`) that the incoming data is a JSON string and that it should be parsed back into a JSON object for use in the route handler. Without this header, the server might not know how to interpret the request body, leading to errors or `req.body` being undefined.

```javascriptfetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
    // This header is crucial for the server to understand the body
```

```
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "foo", body: "bar", userId: 1 })
})
.then(res => res.json())
.then(data => console.log(data));
```

❓ Q #23 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

What is delegated authorization, and how does OAuth facilitate it? Provide a real-world example.

✅ Answer:

Delegated authorization is a process where a user grants a third-party application limited access to their resources on another service, without sharing their actual credentials (username and password) with that third-party application.

OAuth is the protocol that standardizes this process. It provides a secure flow for this delegation.

A common real-world example is using "Login with Google" on a new website (e.g., `example.com`):

1. You click "Login with Google" on `example.com`.
2. `example.com` redirects you to Google's authentication page.
3. You log in directly with Google, using your Google credentials. Google never shares these with `example.com`.
4. Google asks you to consent to `example.com` accessing certain parts of your profile (like your name and email).
5. After you consent, Google redirects you back to `example.com`, providing an **access token**.
6. `example.com` can now use this token to make authorized requests to Google's API on your behalf to get your profile information. You have successfully "delegated" access to `example.com` without ever giving it your password.

❓ Q #24 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🔥 EXTREMELY IMPORTANT

🧠 Question:

What are over-fetching and under-fetching in the context of APIs, and how does GraphQL address these issues?

✅ Answer:

These are common problems with traditional REST APIs:

- **Over-fetching** is when an API endpoint returns more data than the client actually needs. For example, a `/users/1` endpoint might return a full user object with 20 fields, but the UI only needs to display the user's name and avatar. This wastes bandwidth and can slow down the application.
- **Under-fetching** is when an endpoint doesn't return enough data, forcing the client to make additional API calls to fetch everything it needs. For example, to display a user's name and the titles of their last 5 posts, you might have to call `/users/1` and then `/users/1/posts`, resulting in multiple network round trips.

GraphQL solves both problems by allowing the client to specify exactly what data it needs in a single query. The client can request just the `name` and `avatar` to prevent over-fetching, or it can request the user's `name` and their nested `posts`' `titles` in one go to prevent under-fetching.

❓ Q #25 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

How would you configure CORS in an Express backend to specifically allow requests from your React development server running on `http://localhost:5173`?

✅ Answer:

To securely configure CORS for a specific origin, you should use the `cors` middleware package in Express and pass a configuration object to it. Instead of allowing all origins with `app.use(cors())`, which is insecure for production, you should specify the allowed origin in the `origin` property.

This ensures that only your React application running at that specific address can make requests to the backend, while all other origins will be blocked by the

browser's same-origin policy. You can also specify allowed methods and headers for tighter security.

```
javascriptconst express = require("express");
const cors = require("cors");
const app = express();

// CORS configuration object
const corsOptions = {
  origin: 'http://localhost:5173',  // Allow only this origin
  methods: ['GET', 'POST'],  // Allow only these methods
  allowedHeaders: ['Content-Type', 'Authorization']  // Allow only these headers
};

app.use(cors(corsOptions));

// ... rest of the server setup ...
```

❓ Q #26 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is REST (Representational State Transfer)?

✅ Answer:

REST is not a library or a framework, but an **architectural style** for designing networked applications and APIs. A RESTful API is one that adheres to REST principles, exposing resources (data) via a set of predictable endpoints. It uses standard HTTP methods to perform CRUD (Create, Read, Update, Delete) operations.

Method	Action	Example
GET	Read	`/users/5`
POST	Create	`/users`
PUT	Replace	`/users/5`
PATCH	Update	`/users/5`
DELETE	Remove	`/users/5`

❓ Q #27 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture, ⚙️ Performance

🧠 Question:

How does caching differ between REST and GraphQL?

✅ Answer:

The caching strategies for REST and GraphQL are fundamentally different:

- **REST**: Leverages standard, built-in HTTP caching mechanisms very effectively. Because `GET` requests for a specific URL (e.g., `/users/1`) are idempotent and cachable, browsers, CDNs, and proxy servers can easily cache the responses, leading to significant performance gains without client-side effort.
- **GraphQL**: Caching is more complex. Since most GraphQL requests are sent as `POST` requests to a single endpoint, standard HTTP caching is not effective. Instead, caching is primarily handled on the client side, typically within a library like Apollo Client, which uses an in-memory, normalized cache (`InMemoryCache`) to store and serve previously fetched data.

❓ Q #28 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is the purpose of the `express.json()` middleware in an Express server?

✅ Answer:

The `express.json()` middleware is a built-in function in Express that parses incoming requests with JSON payloads. When a frontend application sends a `POST` or `PUT` request with a `Content-Type: 'application/json'` header, this middleware intercepts the request, parses the JSON string from the request body, and attaches the resulting JavaScript object to `req.body`. This makes it easy to access the data sent from the client in your route handlers.

```javascript
// In your server.js
const express = require("express");
const app = express();

// This middleware parses JSON bodies
app.use(express.json());

// Now you can access the parsed data in req.body
app.post("/users", (req, res) => {
  const { name } = req.body;  // { name: "Alice" }// ... create new user
});
```

❓ Q #29 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

How do you define a schema in GraphQL, and what are its main components?

✅ Answer:

A GraphQL schema defines the API's capabilities and acts as a contract between the frontend and backend. It is defined using the GraphQL Schema Definition Language (SDL).

The main components are:

1. **Types**: These define the shape of your data objects. For example, a `User` type would define fields like `id`, `name`, and `email`. The `!` indicates a non-nullable field.
2. **Query Type**: This is the main entry point for read operations. It defines the queries that clients can execute, like fetching a single `user` by ID or a list of all `users`.
3. **Mutation Type**: This is the main entry point for write operations. It defines the mutations clients can execute to create, update, or delete data, like `createUser`.

```graphql
// Example Schema Definition
type Post {
  id: ID!
  title: String!
}

type User {
  id: ID!
  name: String!
  posts: [Post]
}

# Entry points for reading data
type Query {
  user(id: ID!): User
  users: [User]
}

# Entry points for writing data
type Mutation {
  createUser(name: String!): User
}
```

❓ Q #30 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

What are resolvers in a GraphQL implementation?

✅ Answer:

Resolvers are the functions that provide the instructions for turning a GraphQL operation into data. They are the "how" to the schema's "what." For every field in your `Query` and `Mutation` types, you must have a corresponding resolver function that tells the server how to fetch the data for that field.

When a query comes in, the GraphQL server calls the resolver function for the top-level field in the query. That resolver fetches its data, and then the server calls the resolvers for the nested fields, passing the parent's data down.

```javascript
// The root object contains resolvers matching the schema's Query and Mutation fields
const root = {
  // Resolver for the 'users' query
  users: () => {
    return users_data;  // Fetches all users from a data source
  },
  // Resolver for the 'user' query
  user: ({ id }) => {
    return users_data.find(u => u.id == id);  // Finds a user by ID
  },
  // Resolver for the 'createUser' mutation
  createUser: ({ name }) => {
    const newUser = { id: users.length + 1, name };
    users_data.push(newUser);
    return newUser;
  }
};
```

❓ Q #31 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

Explain the concept of a field-level resolver in GraphQL and when you would use one.

✅ Answer:

A field-level resolver is a resolver function that is responsible for fetching the data for a specific field within a type, rather than for a top-level query. This is most commonly used for resolving relationships between types (nested data).

You would use one when a field's data is not directly available on the parent object and needs to be fetched separately. For instance, in a schema where a `User` has `posts`, the top-level `user` resolver might return the user's `id` and `name`. When GraphQL then needs to resolve the `posts` field for that user, it will look for a `posts` resolver on the `User` type. This resolver receives the parent `user` object and can use its `id` to fetch the corresponding posts.

```
javascriptconst root = {
  // Top-level resolver for a single user
 user: ({ id }) => users.find(u => u.id == id),

  // Field-level resolver specifically for the 'posts' field on the User type
 User: {
  posts: (parent) => {
    // 'parent' is the user object returned from the resolver above// Use the parent's ID to find their posts
   return posts.filter(p => p.userId === parent.id);
  }
 }
};
```

❓ Q #32 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is the function call that actually starts an Express server, and what command do you run to start the server file?

✅ Answer:

The function that starts the server is **`app.listen()`**. This method binds the application to a specified port on the host machine and starts listening for incoming connections.

To run the server file (e.g., `server.js`) from your terminal, you use the **`node`** command:

```
bashnode server.js
```

This executes the JavaScript file using the Node.js runtime, and the `app.listen()` call within it will start the server.

```
javascriptconst express = require("express");
const app = express();
const PORT = 3000;

// ... routes and middleware ...// This line actually starts the server
app.listen(PORT, () => {
 console.log(`Server running on http://localhost:${PORT}`);
});
```

❓ Q #33 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 📦 State Management, ⚙️ Performance

🧠 Question:

What is the purpose of Apollo Client's `InMemoryCache`?

✅ Answer:

The `InMemoryCache` is one of the most powerful features of Apollo Client. Its primary purpose is to store the results of your GraphQL queries locally in the browser's memory. This provides several key benefits:

1. **Avoids Redundant Network Requests**: If you request the same data more than once, Apollo Client can serve it directly from the cache instead of making another network call, which significantly improves performance.
2. **Faster UI Updates**: It makes the UI feel instantaneous. When components mount, data that is already in the cache can be rendered immediately while a background fetch updates it if necessary.
3. **State Synchronization**: Apollo Client normalizes the cached data, meaning it can update all parts of your UI that depend on a piece of data when that data changes, ensuring consistency across your application.

❓ Q #34 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 🌐 Routing, 🏗️ Architecture

🧠 Question:

How do you attach an authentication token to every GraphQL request using Apollo Client?

✅ Answer:

You attach an authentication token to every request by creating an "Apollo Link" chain. Specifically, you use `setContext` from `@apollo/client/link/context` to create a middleware link (`authLink`) that intercepts requests before they are sent.

This `authLink` function retrieves the token from its storage location (e.g., `localStorage` or a cookie) and sets the `Authorization` header. You then chain this link with the `httpLink` (which points to your GraphQL endpoint) using `authLink.concat(httpLink)`. This ensures the header is added to every single request made by Apollo Client.

```
 javascriptimport { ApolloClient, createHttpLink, InMemoryCache } from "@apollo/client";
import { setContext } from "@apollo/client/link/context";

const httpLink = createHttpLink({
  uri: 'https://api.github.com/graphql',
});

const authLink = setContext((_, { headers }) => {
    // Get the authentication token from storage
  const token = localStorage.getItem('authToken');
    // Return the headers to the context so httpLink can read them
  return {
    headers: {
      ...headers,
      authorization: token ? `Bearer ${token}` : "",
    }
  }
});

const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache: new InMemoryCache()
});
```

❓ Q #35 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

Describe the complete frontend flow for handling token expiration and refresh.

✅ Answer:

A robust token refresh flow on the frontend involves these steps:

1. **Store Both Tokens**: After login, securely store both the short-lived **access token** and the long-lived **refresh token**. The access token can be in memory, while the refresh token should be in a more secure location like an `HttpOnly` cookie.
2. **Proactive Check (Optional but good)**: Before making an API call, you can decode the access token on the client side to check its `exp` (expiration) timestamp. If it's expired, you can trigger the refresh flow immediately without waiting for a failed API call.
3. **Reactive Handling (Essential)**: When an API request is made with an expired access token, the server should respond with a `401 Unauthorized` status.

4. **Trigger Refresh**: Your API client (e.g., using an Axios interceptor) should catch this `401` error. It should then pause any other pending requests and send the refresh token to a dedicated `/refresh` endpoint.
5. **Receive New Token**: If the refresh token is valid, the server returns a new access token.
6. **Update and Retry**: The frontend updates its stored access token with the new one and retries the original API request that failed. It also resumes any other paused requests.
7. **Handle Refresh Failure**: If the refresh token is also invalid or expired, the flow fails. The application should clear all authentication data and redirect the user to the login page.

❓ Q #36 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What is PostgreSQL?

✅ Answer:

PostgreSQL, often called "Postgres," is a powerful, open-source, and highly stable **relational database management system (RDBMS)**. As a relational database, it stores data in a structured format using tables, which consist of rows (records) and columns (fields). It's known for its reliability, feature robustness, and strong adherence to SQL standards. In a full-stack application, it serves as the persistent "storage room" for all application data.

❓ Q #37 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What role does Prisma play when working with a PostgreSQL database, and what is its main advantage in a TypeScript project?

✅ Answer:

Prisma acts as an **Object-Relational Mapper (ORM)**, creating a high-level abstraction layer between your application code (in Node.js/TypeScript) and your PostgreSQL database. Instead of writing raw SQL queries, you interact

with your database using intuitive, auto-generated JavaScript or TypeScript methods.

Its main advantage in a TypeScript project is **full type safety**. Prisma generates a client based on your database schema (`schema.prisma`), which means your database queries are fully typed. This provides benefits like:

- **Autocompletion** for model and field names in your editor.
- **Compile-time error checking**, which catches typos or incorrect query structures before you even run your code, drastically reducing runtime bugs.

❓ Q #38 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What does the `npx prisma migrate dev` command do?

✅ Answer:

The `npx prisma migrate dev` command is a crucial part of Prisma's workflow for keeping your database schema and your Prisma schema synchronized. When you run it, Prisma performs several actions:

1. It compares your current `schema.prisma` file against the actual state of your database.
2. If there are differences, it automatically generates a new SQL migration file in the `prisma/migrations` directory. This file contains the `ALTER TABLE`, `CREATE TABLE`, etc., commands needed to update the database.
3. It applies this new migration to your development database, updating its structure.
4. It ensures your Prisma Client is regenerated to reflect the latest schema changes.

❓ Q #39 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture, 🧪 Testing

🧠 Question:

What is the purpose of the `npx prisma db seed` command?

✅ Answer:

The `npx prisma db seed` command is used to **populate your database with initial data**. It executes a predefined seed script (e.g., `prisma/seed.ts`). This is extremely useful for development and testing because it ensures you have a consistent set of sample data to work with, such as creating a few default users, products, or configuration entries. This avoids having to manually create data every time you reset your database.

```json
// In package.json, you configure the command to run your seed script
"prisma": {
  "seed": "ts-node prisma/seed.ts"
}
```

❓ Q #40 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

How do you install Express.js in a Node.js project?

✅ Answer:

You install Express.js using the Node Package Manager (npm) with the following command in your project's terminal:

```bashnpm install express
```

This command downloads the Express.js library from the npm registry and adds it as a dependency to your project's `package.json` file.

❓ Q #41 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture

🧠 Question:

Explain how you can implement role-based authorization in an Express.js backend by chaining middleware.

✅ Answer:

Role-based authorization can be implemented cleanly by creating a chain of middleware functions. The flow is typically:

1. **Authentication Middleware**: First, an `authenticate` middleware runs to verify the user's JWT. If the token is valid, it decodes the payload (which should

contain user information, including their `role`) and attaches it to the `req` object (e.g., `req.user`).

2. **Authorization Middleware**: Next, you create a higher-order function, `authorizeRole`, that accepts a `role` (e.g., 'admin'). This function returns a middleware that checks if `req.user.role` matches the required role. If it matches, it calls `next()` to pass control to the next function in the chain (the route handler). If not, it sends a `403 Forbidden` response.

This allows for a declarative and reusable way to protect routes.

```
javascriptfunction authorizeRole(role) {
 return (req, res, next) => {
    // Assumes 'authenticate' middleware has already run and attached req.user
   if (req.user && req.user.role === role) {
     next();   // User has the required role, proceed
   } else {
     res.status(403).json({ message: "Forbidden: Insufficient rights" });
   }
 };
}

 // Chaining middleware: runs authenticate, then authorizeRole('admin')
app.get('/admin', authenticate, authorizeRole('admin'), (req, res) => {
 res.send('Welcome, admin!');
});
```

❓ Q #42 (🟢 Basic)

Tags: 🧩 General Interview, 🔧 Hooks, 💥 Pitfall

🧠 Question:

What is the `signal` property of an `AbortController`, and what is it used for?

✅ Answer:

The `signal` property of an `AbortController` instance is an `AbortSignal` object. It acts as the communication channel between the controller and the operation you want to cancel (like a `fetch` request).

You pass the `signal` into the `fetch` options. The `fetch` operation then "listens" to this signal. When you call `controller.abort()`, the controller sends an "abort" signal through this channel, which `fetch` receives, causing it to immediately terminate the HTTP request and reject the promise with an `AbortError`.

❓ Q #43 (🟡 Intermediate)

Tags: 🧩 General Interview, ⚙️ Performance, 💥 Pitfall

🧠 Question:

How could you automatically cancel a `fetch` request when a user switches to a different browser tab, and why might this be useful?

✅ Answer:

You can achieve this by listening to the `visibilitychange` event on the `document`. This event fires whenever the content of a tab becomes visible or hidden. Inside the event handler, you can check the `document.hidden` property. If it's `true`, you call `controller.abort()`.

This is useful for saving system resources and preventing unnecessary background processing. If a user switches tabs, they are not actively viewing the content, so completing a large data fetch may be wasteful. Canceling it frees up network bandwidth and reduces server load.

```
javascriptconst controller = new AbortController();
fetch(url, { signal: controller.signal });

document.addEventListener("visibilitychange", () => {
  if (document.hidden) {
    controller.abort();
    console.log("Aborted because tab changed");
  }
});
```

❓ Q #44 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 📦 State Management, ⚙️ Performance

🧠 Question:

Compare how `fetch` and Apollo Client handle caching.

✅ Answer:

The approaches to caching are vastly different and highlight a key advantage of using a dedicated client like Apollo.

- `fetch`: Has no built-in data caching mechanism. Every time you call `fetch`, a new network request is made. If you need caching, you must implement it yourself, either by storing responses in a state management solution (like Redux or Zustand) or by using an external data-fetching library (like TanStack Query) that adds a caching layer.

- **Apollo Client**: Has a powerful, automatic, and normalized cache (`InMemoryCache`) at its core. When you make a query, Apollo stores the result in its cache. The next time the same query is made, the data is served instantly from the cache. Because the cache is normalized, if a mutation updates an item, Apollo can automatically update that item everywhere it appears in your UI, ensuring data consistency.

❓ Q #45 (🟡 Intermediate)

Tags: 🚀 MAANG-level, 📦 State Management, 💥 Pitfall

🧠 Question:

Compare how `fetch` and Apollo Client handle the state management related to a data request.

✅ Answer:

The difference illustrates a shift from imperative to declarative data fetching.

- `fetch`: Requires manual, imperative state management. For every request, you need to use `useState` to manually track `loading`, `error`, and `data` states. You must remember to set `setLoading(true)` before the request, `setData(result)` on success, `setError(e)` on failure, and `setLoading(false)` in both completion scenarios. This is boilerplate-heavy and prone to errors.
- **Apollo Client**: Handles state management declaratively and automatically via the `useQuery` hook. The hook returns an object containing `loading`, `error`, and `data` properties. These properties are automatically updated by Apollo throughout the request's lifecycle. Your component code becomes much cleaner, as it only needs to react to these states rather than manage them.

```jsx
// Manual state management with fetch
const [data, setData] = useState(null);
const [loading, setLoading] = useState(true);
useEffect(() => {
  setLoading(true);
  fetch(url).then(res => res.json()).then(setData).finally(() => setLoading(false));
}, []);

// Declarative state management with Apollo
const { loading, error, data } = useQuery(GET_DATA);
// No manual state setting is needed.
```

❓ Q #46 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture, 🧰 Debugging

🧠 Question:

What is the `graphiql: true` option used for when setting up a GraphQL endpoint with `express-graphql`?

✅ Answer:

Setting the `graphiql: true` option in the `graphqlHTTP` middleware enables the **GraphiQL interface**. This is a powerful, in-browser IDE for your GraphQL API that is served directly from your GraphQL endpoint. It is an invaluable development tool that allows you to:

- Write and test queries and mutations interactively.
- Explore your full API schema through its documentation explorer.
- Get autocompletion and real-time error highlighting for your queries.

❓ Q #47 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

What is a "Man-in-the-Middle" (MITM) attack, and what is the primary way to prevent it when transmitting authentication tokens?

✅ Answer:

A **Man-in-the-Middle (MITM)** attack is a type of eavesdropping where an attacker secretly intercepts and relays communication between two parties who believe they are communicating directly with each other. If the communication is unencrypted, the attacker can read and even modify the data being exchanged, such as stealing an authentication token.

The primary and most critical defense against MITM attacks is to **always use HTTPS** for all communication between the client and server. HTTPS encrypts the entire data stream using TLS/SSL, making it impossible for an attacker on the network to read the content of the requests or responses, thus protecting the transmitted tokens.

❓ Q #48 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, 💥 Pitfall

🧠 Question:

Why is storing sensitive information like authentication tokens in URL parameters a severe security risk?

✅ Answer:

Storing tokens in URL parameters is a severe security risk because URLs are highly visible and are not treated as secure storage locations. They are frequently logged, cached, and shared in plain text in numerous places, including:

- **Browser History**: The full URL, including the token, is stored in the user's browser history.
- **Server Logs**: Web servers typically log the full URL of every incoming request.
- **Network Proxies and Caches**: Intermediate network devices can log URLs.
- **Referrer Headers**: If the user clicks a link to an external site, the browser may send the full URL (with the token) in the `Referer` header.

  This widespread exposure makes it trivial for an attacker with access to any of these logs to steal the token.

❓ Q #49 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

Explain the purpose of the `DATABASE_URL` environment variable in a project that uses Prisma and PostgreSQL.

✅ Answer:

The `DATABASE_URL` is a **connection string** that provides all the necessary information for your application (via Prisma) to connect to your PostgreSQL database. It is stored as an environment variable for security and flexibility, separating configuration from code.

It consolidates all connection details into a single string, typically in the format:

`postgresql://USER:PASSWORD@HOST:PORT/DATABASE`

Prisma reads this variable from the `.env` file to know exactly which database server to connect to, what credentials to use, and which specific database to

interact with.

❓ Q #50 (🟡 Intermediate)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

How do you define a one-to-many relationship between two models, such as `User` and `Product`, in a `schema.prisma` file?

✅ Answer:

You define a one-to-many relationship by linking the two models on both sides. In this case, one `User` can have many `Product`s.

1. **On the "one" side (`User`)**: You add a field that represents the list of related models. This field has the type of the other model as an array (e.g., `products Product[]`).
2. **On the "many" side (`Product`)**: You add two fields:
   - A field that holds a single instance of the related model (e.g., `user User`). You use the `@relation` attribute here to define the relationship, specifying which fields to use for the join (`fields: [userId]`) and what they reference (`references: [id]`).
   - A foreign key field that will store the ID of the related `User` (e.g., `userId Int`).

```
textmodel User {
 id      Int      @id @default(autoincrement())
 email   String   @unique
 // A user can have many products
 products Product[]
}

model Product {
 id     Int    @id @default(autoincrement())
 name   String
 // Link back to the User model
 user   User   @relation(fields: [userId], references: [id])
 userId Int    // Foreign key
}
```

❓ Q #51 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 Question:

What are the common HTTP methods used in a RESTful API, and what CRUD operations do they typically map to?

✅ Answer:

The common HTTP methods and their corresponding CRUD (Create, Read, Update, Delete) operations are:

- **GET**: **Read** - Used to retrieve a resource or a collection of resources.
- **POST**: **Create** - Used to create a new resource.
- **PUT**: **Update/Replace** - Used to completely replace an existing resource at a specific URI.
- **PATCH**: **Update/Modify** - Used to apply partial modifications to a resource.
- **DELETE**: **Delete** - Used to remove a resource.

❓ Q #52 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🔧 Hooks, 💥 Pitfall

🧠 Question:

How do you handle request cancellation in Axios, and how does the error handling differ from `fetch`?

✅ Answer:

You can cancel an Axios request using the same `AbortController` API as `fetch`. You create a controller and pass its `signal` in the Axios request configuration object.

The key difference is in error handling.

- With `fetch`, a canceled request rejects the promise with an error whose `name` property is `'AbortError'`. You check for this specific error name.
- With **Axios**, the library has its own way of identifying cancellation errors. Instead of checking the error name directly, the recommended approach is to use the `axios.isCancel(err)` type guard function. This function returns `true` if the error was due to a request cancellation, providing a more reliable and library-specific way to handle the error.

```javascript
// Using Axios with AbortController
const controller = new AbortController();

axios.get(url, { signal: controller.signal })
```

```
  .catch(err => {
     // Use the axios helper to check for cancellation
    if (axios.isCancel(err)) {
      console.log("Axios request canceled");
    } else {
       // Handle other errors
    }
  });

 // Cancel the request
controller.abort();
```