Handling Cookies in the frontend

These code snippets illustrate the interaction between server-side and client-side code in a **Next.js** application using cookies.
We will use both next/headers for cookies in server components and cookies-next for cookies in client components

Here's a detailed breakdown:

Code 1: `Login` Component (server-Side)

Overview of the Login Server Action

The provided login code is a **server action** in Next.js, specifically written to handle a **login request**. It interacts with an external backend to authenticate the user, sets cookies based on the server response, and redirects the user upon successful login.

Let's break down the code and explain how it works:

Code Breakdown:

1. `'use server'`:
   - This directive marks the function as a **server-side action** in Next.js, which runs on the server and doesn't directly interact with the client-side state (unlike client-side components).
2. `axiosInstance`:
   - This is an **Axios instance** configured to send HTTP requests to the backend server (`http://localhost:3001` in this case).
   - `axiosInstance` is used to make a POST request to the `/auth/login` endpoint with the user's login credentials (`email` and `password`).
3. `cookies()` **(from Next.js):**
   - The `cookies()` function from `next/headers` allows the server to interact with cookies, which are typically used for session management or state persistence between the client and server.
4. `axiosInstance.post`:
   - The `POST` request sends the login credentials (email and password) to the backend for authentication.
5. **Response Handling:**
   - If the login is successful (i.e., status code `201`), the server extracts the **authentication token** from the `set-cookie` header.

    - The `set-cookie` header contains information for setting cookies in the client's browser. The token is stored as `CookieFromServer`, with security flags (e.g., `secure`, `httpOnly`, `sameSite`).
    - The **maxAge** (expiration time) is extracted from the `set-cookie` header to ensure the cookie is valid for a certain period.
6. **Cookie Storage:**
    - After receiving the token and maxAge from the backend, the server sets the cookie `CookieFromServer` with the token and expiration time using `cookieStrore.set()`.
    - This cookie is **secure**, **HTTP-only**, and uses the **sameSite** policy to protect it from cross-site scripting and other potential vulnerabilities.
7. **Redirection:**
    - After successfully setting the cookie, the user is redirected to the `/about` page using `redirect('/about')`.
8. **Error Handling:**
    - If the login request fails (e.g., wrong credentials), the error message is captured and returned as part of the response.

Code 2: `Login(Page.tsx)` Component (Client-Side)

Explanation of the `page.tsx` for the Login Route

The provided `page.tsx` code is for the **login page** in a Next.js application. It functions as a client-side component that allows users to enter their credentials, submit the login form, and trigger the `login` server action. Here's a breakdown of how this code works and how it fits into the context of the overall application:

Code Breakdown:

1. **`'use client'`:**
    - This directive indicates that the code runs on the client side, enabling the use of React hooks, browser APIs, and client-side interactivity.
2. **Imports:**
    - **`useActionState` and `useState` from React**: `useActionState` is a Next.js hook used to handle the form action state, which helps manage form submissions. `useState` is used to manage the local state of the email and password inputs.
    - **`useRouter` from `next/navigation`**: Provides programmatic navigation capabilities. It allows the page to redirect after a successful login.

- **`axiosInstance`**: An Axios instance for making HTTP requests to the backend (though not used directly in this component, it might be imported for potential future use or consistent request headers).
- **`login` function from `./login.server`**: The server action that handles the login logic, which was explained earlier.

3. **State Management:**
   - **`const [email, setEmail] = useState<string>("");`**
     - State variable to store the user's email input.
   - **`const [password, setPassword] = useState<string>("");`**
     - State variable to store the user's password input.
   - **`const [state, formAction] = useActionState(login, { message: '' });`**
     - `useActionState` initializes the server action (`login`) with a default `message` state and returns `state` (the current action state) and `formAction` (the function to handle the form submission).
4. **Form Structure:**
   - The form has the `action` attribute set to `formAction`, making it submit data to the `login` server action when submitted.
   - **Input Fields:**
     - **Email Input:**
       - `type="email"` specifies the input type.
       - `name="email"` ensures that this field is submitted as part of the `formData` to the server action.
       - `value={email}` and `onChange={(e) => setEmail(e.target.value)}` link the input value to the `email` state and update it on change.
     - **Password Input:**
       - `type="password"` ensures the password is hidden during input.
       - `name="password"` is the field name used in the form data.
       - `value={password}` and `onChange={(e) => setPassword(e.target.value)}` link the input value to the `password` state and update it on change.
   - **Submit Button:**
     - `type="submit"` triggers the form submission, invoking the `login` server action.
5. **Display State Messages:**
   - The `<p>{state?.message}</p>` displays a message based on the `state` returned from the `login` action. If there's an error or success message, it will be shown to the user

---

Code 3: `About` Component (Server-Side)

This is a server-side component defined in a file like `about/page.js` in a Next.js application.

Key Points:

1. **`import { cookies } from 'next/headers'`:**
   - The `cookies()` function is a Next.js API that allows access to HTTP cookies on the server side.
2. **`cookies().get('CookieFromServer')?.value`:**
   - Retrieves the value of the `CookieFromServer` cookie from the HTTP request.
   - If the cookie exists, its value is fetched; otherwise, `undefined` is returned.
3. **`console.log((await cookies()).get('CookieFromServer'))`:**
   - Logs the retrieved cookie value to the server console for debugging.
4. **`<MyClientComponent>`:**
   - A React client-side component is rendered, and the cookie value is passed as the `initial` prop.

Flow:

- The server-side `About` function fetches the cookie from the HTTP request and passes it to the client-side `MyClientComponent` as part of its `initial` prop.

Code 4: `MyClientComponent` (Client-Side)

This is a client-side component that interacts with cookies and renders the UI.

Key Points:

1. **`'use client'`:**
   - Marks this component as a client-side component in Next.js.
2. **`import { getCookie, setCookie } from 'cookies-next'`:**
   - `cookies-next` is a package for managing cookies in a browser environment.
   - `getCookie`: Reads cookies on the client side.
   - `setCookie`: Sets cookies on the client side.
3. **`initial.cookieClient` (prop):**
   - The `initial` prop provides the cookie value fetched server-side as a fallback.
4. **`useState`:**
   - Initializes `cookieClient` state with one of the following:

        - The client-side cookie value (`getCookie('CookieFromServer')`).
        - The server-side cookie fallback (`initial.cookieClient`).
        - An empty string if neither is set.
5. **`useEffect`:**
    - Synchronizes the `cookieClient` state with a new cookie (`cookieClient`).
    - The `maxAge` property ensures the cookie expires after a set time (3600 seconds here).
6. **JSX Output:**
    - Renders:
        - A message: "Hi from About".
        - The current cookie value (`cookieClient`).
        - A link to another page (`/about/me`).

Key Functionalities and Flow:

1. **Server-Side:**
    - The server extracts the `CookieFromServer` cookie from the HTTP request and passes it to the client-side.
2. **Client-Side:**
    - The client initializes with the cookie value fetched from the server, updates it dynamically, and sets a new cookie.
3. **Dynamic Updates:**
    - If `cookieClient` changes (due to user interaction or another effect), the updated value is written as a new cookie using `setCookie`.
4. **Integration:**
    - This pattern is useful in applications requiring cookies for session management, personalization, or sharing state between server-side and client-side code.

Example Scenario:

- A user visits the `/about` page.
- The server-side `About` function retrieves and passes a cookie to `MyClientComponent`.
- On the client side:
    - The cookie is displayed in the UI.
    - Any changes to the `cookieClient` state are written back as a cookie.

This demonstrates seamless integration of server-side and client-side logic with cookies in a Next.js app.

Code 5: `About/Me` Component (Client-Side)

The `/about/me` page is a **client-side component** in a Next.js application that interacts with cookies using the `cookies-next` library. Here's a detailed breakdown:

Code Explanation:

Key Components:

1. `'use client';`:
   - Marks this component as a **client-side component** in Next.js.
   - Client-side components can use React hooks (like `useState` and `useEffect`) and directly interact with the browser (e.g., for cookies or local storage).
2. `getCookie` **from** `cookies-next`:
   - Reads the value of the cookie named `cookieClient` on the client-side (i.e., in the browser).
3. `useState`:
   - Initializes the `cookieClient` state with the value fetched by `getCookie("cookieClient")`.
   - If the cookie doesn't exist, the state defaults to an empty string (`""`).
4. **JSX Output:**
   - Displays:
     - A greeting message.
     - The value of the `cookieClient` cookie in a `<p>` tag.

Behavior and Flow:

1. **State Initialization:**
   - When the page loads, `getCookie("cookieClient")` is called to fetch the cookie value.
   - This value is used to initialize the `cookieClient` state.
2. **No** `useEffect`:
   - Unlike the previous component (`MyClientComponent`), this component doesn't dynamically update or set cookies using `useEffect`.
   - It is designed as a **read-only page** to display the cookie value.
3. **Static Display:**
   - The cookie value is displayed immediately without any interaction required.

How Page Relates to each others:

Interaction with the Server Action (`login`):

- When a user submits the form, the `login` server action is invoked, taking the email and password submitted through `formData`.
- The `login` action verifies the user credentials on the backend, sets a cookie (`CookieFromServer`) if successful, and redirects to the `/about` page. If the login fails, an error message is returned and displayed on the page.

Cookies and Authentication:

- The `login` server action sets the `CookieFromServer` cookie in the response header after a successful login. This cookie is then available to other pages (e.g., `/about`, `/about/me`) for authentication and session management.
- The cookie is then read by server-side (`About` page) and client-side (`MyClientComponent` and `/about/me`) components, as explained in the previous code breakdowns.

Form Submission Flow:

1. The user enters their **email** and **password** in the login form and submits it.
2. The form triggers the **`login` server action** through `useActionState`, which processes the login and sets the cookie if successful.
3. The user is redirected to the `/about` page upon a successful login, where they can view or interact with the app based on their authenticated state.

Summary of the Full Flow:

1. **Login Page (`page.tsx`):**
    - Provides a UI for users to input their email and password and submit the form.
    - Displays any messages from the login process, such as errors or confirmation of login.
2. **Login Server Action (`login.server.ts`):**
    - Handles the authentication request, sets cookies, and redirects the user based on the outcome.
3. **Authenticated Pages (`/about`, `/about/me`):**
    - Read and display the authentication token from cookies, showing that the user is logged in and maintaining session state.