Adi’s feedback

General Feedback:

You clearly have a good foundation in both frontend and backend development. However, there were a few bugs and areas where improvements could be made to enhance the quality, maintainability, and scalability of the code.

Frontend:

1. **Random User Initialization Bug:**
   - A significant bug was the random initialization of the user using Math.ceil(Math.random() * 4). This approach could lead to unpredictable behavior since the user is randomly set and never correctly updated. This leads to loading the data of a random user with no relation to the set user in the URL. When choosing to hide offers/affiliations for a certain user and reloading the page we get the offers of another user.
   - **Suggested Fix:** Instead of randomly initializing the user, the user should be fetched from an authentication source (e.g., a logged-in user's data) or set explicitly through the URL param.
2. **Type Safety:**
   - Using **any** types in your components (e.g., for action in SuggestedActionItem.tsx) reduces the benefits of TypeScript. It’s important to define explicit types, as this helps catch potential errors early and improves readability.
3. **Error Handling:**
   - A notable issue was the incomplete error handling in the hideAffiliant and hideOffer functions within offersService.tsx and how they were called in SuggestedActionsSection.tsx. While you included try/catch blocks, errors were logged to the console but not propagated or handled effectively, leaving the user uninformed when something went wrong. In the SuggestedActionsSection.tsx component, if the hide operation failed, there was no feedback for the user. This can create a poor user experience, as users wouldn’t know if their action succeeded or failed.
   - **Suggested Fix:** In addition to logging errors, consider updating the UI to reflect the error (e.g., setting an error state or displaying a message). Additionally, always ensure that the UI is updated correctly, even after an error.
4. **State Management and Loading:**
   - The loading state was not handled correctly. Since setLoading(false) wasn't placed in a finally block, the UI could get stuck in a loading state if an error occurred.
   - **Suggested Fix:** Place setLoading(false) in a finally block to ensure it runs regardless of whether the operation succeeds or fails.

5. **Keys in Lists:**
    - In the map function for rendering offers, object references were used for the key prop. This can cause issues with React's rendering behavior and lead to unnecessary re-renders.
    - **Suggested Fix:** Always use a unique identifier (like offerItem.id) for the key prop to ensure React can manage updates correctly and efficiently.
6. **Optimizing User Experience with Async Operations:**
    - A potential race condition exists when hiding offers or affiliates, as the list of offers is refetched after each hide operation. This could lead to out-of-sync UI if multiple operations happen quickly.
    - **Suggested Fix:** Implement optimistic updates to hide the offer immediately on the frontend, and handle the server request in the background.

Backend:

1. **Cache Utilization:**
    - While you introduced a cache object in the OffersService, it wasn't utilized effectively. This leads to unnecessary database queries every time offers are fetched, which can degrade performance, especially as the data scales.
    - **Suggested Fix:** Use caching to store frequently accessed data, such as hidden offers or affiliates, to avoid hitting the database repeatedly. This would significantly improve the app's performance.
2. **Inefficient Query in getAllOffers:**
    - The implementation of the getAllOffers function was inefficient. The logic involves querying the database for hidden offers and affiliants and then fetching all offers, and filtering them in memory. This approach can be expensive, especially as the dataset grows.
    - **Suggested Fix:** Optimize the query by filtering out hidden offers and affiliants directly in the database query itself, rather than fetching everything and filtering in memory. Using a more efficient query structure (e.g., using WHERE NOT IN clauses) will reduce the load on both the server and database.
3. **Error Handling in Services:**
    - Similar to the frontend, error handling in the backend could be improved. In getAllOffers, errors were caught but not properly propagated, meaning the client would never know when something went wrong.
    - **Suggested Fix:** Ensure that errors are not just logged but also communicated back to the client, so the frontend can display appropriate error messages and the user is kept informed.
4. **Database Transactions: (A good tip for working with postgres in larger projects)**

- The lack of transactions when hiding offers or affiliates could lead to inconsistent data. For example, if something fails during the operation, part of the data could be updated while other parts are not, leading to potential issues.
- **Suggested Fix:** Use transactions to ensure that multiple operations (such as hiding offers and updating related records) are executed as a single unit. This guarantees data consistency across the system.