Section 3: Conceptual Questions

1. **How would you ensure a tabs component is responsive for mobile and desktop, considering layout and user interaction?**

   To make a tabs component responsive, I'd start by using Bootstrap's built-in navs and tabs components (.nav, .nav-tabs, .nav-item, etc.). Bootstrap handles a lot of responsiveness out of the box with its grid and flex utilities, so on desktop, the tabs would display horizontally by default.

   For mobile screens, if the tabs start to overflow, I'd use a scrollable tab layout by applying overflow-x: auto via a custom class or Bootstrap utility (overflow-auto, d-flex, etc.), along with flex-nowrap so the tabs stay in one line and can be scrolled horizontally. This works well with touch gestures on mobile.

   Additionally, I'd make sure the tab labels are short and icons are used when helpful. Accessibility is also key, so I'd ensure proper ARIA attributes like role="tablist", aria-selected, and keyboard navigation support are in place.

2. **How would you optimize a virtualized list for 1000+ logs to minimize rendering overhead?**

   When dealing with 1000+ logs, rendering everything at once can really hurt performance especially in React or React Native. So, I'd use a virtualization library like react-window or react-virtualized. These tools render only the visible portion of the list and recycle DOM elements as the user scrolls, which massively reduces rendering overhead.

   If the items have dynamic heights, I'd use something like VariableSizeList from react-window. I'd also memoize each list item with React.memo to avoid unnecessary re-renders, and if the data is fetched from an API, I'd consider pagination or infinite scroll so that we load just a portion at a time.

   In short only render what's visible, and load more as needed.

3. **What accessibility features ensure a dropdown menu and modal are usable for screen readers and keyboard navigation?**

   To make a dropdown menu and modal accessible, I'd focus on both screen reader support and keyboard navigation.
   For screen readers, I'd use ARIA roles and attributes:
   For a dropdown: role="menu", aria-haspopup="true", and aria-expanded to indicate when it's open.
   For modals: role="dialog" or role="alertdialog", and link it to a title using aria-labelledby, and optionally a description using aria-describedby.

   For keyboard users:
   Dropdown items should be reachable using arrow keys, and selectable with Enter or Space.

Modals should trap focus inside meaning Tab and Shift+Tab should loop within the modal elements.
Pressing Escape should close the modal.

When the modal closes, focus should return to the element that opened it.

These small steps really improve usability for users relying on assistive tech.

4. **Describe strategies to secure client-side authentication (e.g., JWT storage) in a wellness app.**

For securing client-side authentication, especially using JWTs, the first thing I'd consider is where and how the token is stored. The most secure option is to store the JWT in an HttpOnly cookie, because it's not accessible via JavaScript, which helps prevent XSS attacks.

If for some reason we need to use localStorage or sessionStorage, I'd make sure to:
Set a strong Content Security Policy (CSP).
Sanitize all inputs to prevent XSS.
Use short-lived access tokens with refresh tokens securely stored and rotated.

Other important strategies include:
Always using HTTPS to encrypt data in transit.
Implementing token expiration and automatic logout when a token becomes invalid.
Adding rate limiting and brute-force protection on authentication endpoints.

secure storage + token lifecycle management + defensive coding = safer client-side authentication.

5. **What are the benefits and challenges of implementing server-side rendering for a wellness dashboard with authentication?**

Benefits:
SSR speeds up the initial page load because the HTML is generated on the server and sent fully formed, so users see content faster.
It's great for SEO if the dashboard has any publicly accessible parts.
SSR can improve performance on slower devices since the heavy rendering is done server-side.

Challenges:
Handling authentication securely is trickier because you have to manage user sessions or tokens on the server as well as the client.
You need to synchronize auth state between server and client, which adds complexity.

SSR can increase server load and infrastructure costs.
Sometimes SSR means extra work setting up data fetching and caching properly to avoid delays.
In short, SSR can make your app feel faster and more discoverable but requires careful handling of auth and server resources.

React Native

1. **What are the differences between View, Text, and ScrollView in React Native?**

   View, Text, and ScrollView are basic building blocks in React Native, but they serve different purposes:

   View is like a container or a box you use it to group other components together or to control layout. Think of it like a in web development.

   Text is specifically for displaying text. Unlike the web, in React Native, you can't just put plain text anywhere; it has to be wrapped inside a Text component.

   ScrollView is a container that allows its children to be scrollable vertically (or horizontally). It renders all its child components at once, so it's best for small to medium amounts of content, but not ideal for very long lists because it can affect performance.

2. **In a react native application, how would you securely store and manage sensitive user credentials such as access tokens and refresh tokens (tools or libraries)?**

   In a React Native app, to securely store sensitive credentials like access and refresh tokens, I'd avoid using plain storage like AsyncStorage because it's not encrypted and vulnerable to attacks.

   Instead, I'd use secure storage libraries such as:

   react-native-keychain: Uses the device's secure storage (Keychain on iOS, Keystore on Android) to encrypt and protect credentials.

   react-native-encrypted-storage: Provides encrypted, persistent storage on the device.

   If using Expo, SecureStore is a good built-in option for encrypted storage.

   Additionally, combining secure storage with biometric authentication (fingerprint, Face ID) adds an extra layer of security.

   Finally, managing token expiration and refresh securely, and never exposing tokens in logs, helps maintain safety.

3. **What's the difference between navigate and push in a stack navigator?**

navigate takes you to a specific screen, but if that screen is already somewhere in the stack, it simply brings you back to it instead of adding a new instance. So, it won't stack duplicates of the same screen.

push, on the other hand, always adds a new instance of the screen on top of the stack even if you're already on that screen. This is useful if you want to show the same screen multiple times with different data.

4. **Explain how to improve the performance of a React Native app.**

Use FlatList or SectionList instead of ScrollView for rendering large lists. These components only render what's visible on the screen, which saves memory and improves speed.

Avoid unnecessary re-renders by using React.memo, useCallback, and useMemo to memoize components and functions so they don't update unless needed.

Optimize images by resizing and compressing them before including them in your app, and use caching strategies.

Keep component trees shallow and avoid heavy computations or expensive operations on the main thread.

Consider moving complex or heavy logic to native modules if needed, or use libraries that optimize rendering and animations.

Use tools like React Native Performance Monitor and Flipper to profile and identify bottlenecks.