1. **When would you choose SSR vs SSG vs ISR in App Router?**
   Answer:

- **SSR**: highly dynamic per-request data (user/session, rapidly changing).
- **SSG**: content rarely changes, best performance.
- **ISR**: mostly static but needs periodic or on-demand updates.

2. **Explain Next.js fetch caching in App Router.**
   Answer: fetch is cached by default in RSC. Control with { cache: "no-store" } (always dynamic) or { next: { revalidate: 60 } } (ISR). Revalidation can be tag-based with revalidateTag() + fetch(..., { next: { tags: ["posts"] }}).

3. **How do you use Server Actions?**
   Answer: Define an async function with the "use server" directive and pass it to forms or call from Client Components:

```
// app/actions.ts

"use server";

export async function savePost(data: FormData) { /* … DB write … */ }
```

They run on the server, removing the need for many API routes.

4. **How to read cookies/headers in App Router?**
   Answer: Use cookies() and headers() from next/headers within Server Components or route handlers.

5. **How to protect a server component route (auth)?**
   Answer: Check session in the server boundary (e.g., using NextAuth or your own cookie/JWT check) inside a layout or page; redirect unauthenticated users with redirect("/login").

6. **What is Middleware and a good use case?**
   Answer: middleware.ts runs before a request is handled (Edge). Use for auth gating, A/B flags, geo-based redirects, or locale detection.

7. **Dynamic Routes & Parallel/Intercepted Routes—when to use them?**
   Answer:

- **Dynamic** ([id]): detail pages.
- **Parallel** (@slot): render multiple route trees at once (e.g., master/detail).

- **Intercepted** ((.) (..)): show a route in a modal over another.

8. **How to stream data to the client?**
   Answer: Use RSC + Suspense in the App Router; Next streams HTML progressively from the server.

9. **How to handle client-only libraries (e.g., charts)?**
   Answer: Mark component with "use client" and optionally dynamic(() => import("./Chart"), { ssr: false }) to avoid SSR issues.

10. **How to do i18n?**
    Answer: Next's built-in i18n routing or libraries like next-intl. Use locale segments or middleware for detection, keep messages server-side for RSC.

11. **How to reduce bundle size?**
    Answer: Prefer Server Components, dynamic imports for large client libs, remove unused MUI icons, analyze with next build + @next/bundle-analyzer.

12. **How to deploy and run in production?**
    Answer: next build then run on Node (or Vercel). For self-host, use output: "standalone" in next.config.js to get minimal artifacts for Docker.

13. **Explain the difference between getStaticProps and getServerSideProps. When would you use one over the other?**

- **A:**
  - **getStaticProps**: Fetches data at **build time**. The page is pre-rendered into static HTML. Use this when the data doesn't change frequently and can be the same for all users (e.g., a blog, documentation).
  - **getServerSideProps**: Fetches data on **each request**. The page is rendered on the server for every user. Use this when the data is dynamic and needs to be up-to-date for every request (e.g., a user's dashboard, real-time stock prices).

14**. What is an API route in Next.js? How is it different from a regular page?**

- **A:** An **API route** is a server-side endpoint created in the pages/api directory. It's used to build your own API, handle form submissions, or fetch data from a database. Unlike a regular page, an API route does not render a UI; it returns data, typically in **JSON** format.

15. **How do you handle state management in a Next.js application? Name a few options.**

- **A:** While Next.js itself doesn't have a built-in state management solution, you can use standard React libraries. Popular options include:

    - **React Context API**: For simple, global state.
    - **Redux, Redux Toolkit, or Zustand**: For more complex, large-scale applications.
    - **Recoil or Jotai**: For atom-based, fine-grained state management.

16. **Describe how Next.js optimizes images using the next/image component. Why is it better than a standard <img> tag?**

- **A:** The next/image component provides a host of optimizations automatically:
    1. **Lazy Loading**: Images are only loaded when they enter the viewport.
    2. **Image Optimization**: Images are automatically sized and formatted (e.g., to WebP) for the user's device and browser.
    3. **Responsive Sizing**: It automatically generates srcset and sizes attributes for different screen sizes.
    4. **Layout Shift Prevention**: It reserves space for the image to prevent layout shifts.

    - It's better than a standard <img> tag because it handles all these performance optimizations for you out of the box, leading to faster page loads and a better user experience.

17. **What is Incremental Static Regeneration (ISR)?**

- **A: ISR** allows you to update static pages without a full rebuild of your site. You specify a revalidate value in getStaticProps. Next.js will then regenerate the page in the background after a certain time interval, serving the stale page while the new one is being built. This combines the performance benefits of **SSG** with the data freshness of **SSR**.

18. **What is Next.js Middleware and what are some common use cases?**

- **A:** Middleware allows you to run code on the server before a request is completed. It runs at the "edge" (on a CDN network) and can intercept the request to rewrite, redirect, add headers, or stream a response. Common use cases include authentication checks, A/B testing, handling bot detection, and managing redirects based on a user's location.

19. **When creating a dynamic page with getStaticProps, what other function is required and why?**

- **A:** You must also export getStaticPaths. Since the page is being generated at build time, Next.js needs to know all the possible dynamic paths that exist beforehand.

getStaticPaths returns an object containing a paths array, which lists all the routes to pre-render (e.g., [{ params: { id: '1' } }, { params: { id: '2' } }]).

20. **How do you handle environment variables in a Next.js application? What's the difference between server-side and client-side variables?**

- **A:** Environment variables are managed using a .env.local file.
  - **Server-side variables:** Any variable (e.g., DB_PASSWORD=secret) is available only on the server-side (in getStaticProps, getServerSideProps, API routes).
  - **Client-side variables:** To expose a variable to the browser, you must prefix it with NEXT_PUBLIC_ (e.g., NEXT_PUBLIC_API_URL=...). This is a security feature to prevent accidentally leaking sensitive keys to the client.

21. **You have a component that is expensive to render and only needed on the client-side (like a complex charting library). How do you prevent it from being included in the server-side render and the initial JavaScript bundle?**

- **A:** You would use **dynamic imports** with the next/dynamic function and disable SSR for that component.

JavaScript

```
import dynamic from 'next/dynamic';


const HeavyChartComponent = dynamic(
  () => import('../components/HeavyChart'),
  { ssr: false }
);
```

This tells Next.js to load the component lazily on the client-side only when it's needed, improving the initial page load performance.

22. **What is the purpose of the fallback key in the object returned by getStaticPaths? Explain the different values (false, true, 'blocking').**

- **A:** The fallback key controls how Next.js behaves when a user requests a path that was not pre-generated at build time.
  - fallback: false: Any path not returned by getStaticPaths will result in a 404 page.

- fallback: true: Next.js will serve a "fallback" version of the page (e.g., a loading spinner) and then try to generate the page in the background on the first request. Subsequent requests for the same path will serve the generated page.
- fallback: 'blocking': The user's browser will "wait" for the page to be generated on the server (SSR) on the first request. There is no fallback state shown. Subsequent requests serve the static version.