1. Basics of Next.js

1. What is Next.js? Why use it?

Next.js is a **production-ready, open-source React framework** that enables developers to build fast, scalable, and SEO-friendly web applications. Built on top of React, it extends React's capabilities by providing a structured, full-stack solution with built-in features like **server-side rendering (SSR)**, **static site generation (SSG)**, and **API routes**.

We use Next.js primarily because it solves many of the challenges associated with building large-scale React applications. It handles crucial aspects like **performance optimization**, **routing**, and **data fetching** out-of-the-box, saving developers time and effort. Its focus on **SEO** and **performance** makes it an excellent choice for modern web development.

2. Difference between Next.js and React.js.

Think of **React.js** as the core library for building user interfaces. It's like the engine of a car—it's powerful, but to have a complete, road-ready vehicle, you need a chassis, wheels, and a body. **Next.js** is that complete car. It's a **framework** built on top of React that provides the necessary infrastructure to build a full-fledged application.

Feature	React.js (without framework)	Next.js (with framework)
Purpose	UI library for creating single-page applications (SPAs)	Full-stack framework for building server-rendered or static-generated websites
Rendering	Client-Side Rendering (CSR)	Multiple rendering methods: **SSR, SSG, ISR, CSR**
Routing	Manual configuration with libraries like React Router	**File-based routing** (automatic)
Data Fetching	Typically manual with `useEffect` or data-fetching libraries	Built-in data-fetching functions like `getServerSideProps` and `getStaticProps`

SEO	Poor out-of-the-box due to CSR (search engines may struggle to crawl)	Excellent due to server-side rendering and static generation
API	No built-in backend/API support; requires a separate server (e.g., Express.js)	Built-in **API routes** for creating backend endpoints
Configuration	Extensive manual configuration (e.g., Webpack, Babel)	**Zero-config** or minimal configuration needed

3. Key features of Next.js (SSR, SSG, ISR, API routes).

- **Server-Side Rendering (SSR)**: Generates the HTML for each page on the server for every request. This is great for data that changes frequently and for which you need up-to-the-minute information. It provides excellent SEO and a fast Time to First Byte (TTFB).
- **Static Site Generation (SSG)**: Generates HTML and JSON files at build time. This is ideal for content that doesn't change often, like blog posts or documentation. The pre-built pages are served from a CDN, making them incredibly fast and resilient.
- **Incremental Static Regeneration (ISR)**: A hybrid of SSR and SSG. It allows you to **statically generate pages** at build time and then **revalidate them on demand** or after a certain period of time without needing a full rebuild. This is perfect for pages that need to be updated but not with every request.
- **API Routes**: Allows you to create API endpoints within your Next.js application. You can use this to build a full-stack application without needing a separate backend server.

4. What is the difference between CSR, SSR, SSG, and ISR?

Method	Description	Best For	Example
CSR (Client-Side Rendering)	HTML is a bare shell; JavaScript fetches and renders all data on the client browser.	Dynamic dashboards, user-specific data.	A logged-in user's profile page.

SSR (Server-Side Rendering)	HTML is fully rendered on the server for each request.	Pages with highly dynamic, frequently changing data.	An e-commerce product page showing real-time stock.
SSG (Static Site Generation)	HTML is generated at build time; pages are pre-built and served as static assets.	Pages with static content (blogs, marketing pages).	A blog post, an 'About Us' page.
ISR (Incremental Static Regeneration)	Pages are statically generated but can be updated after a set time or a revalidation request.	Pages that need to be updated periodically but don't need to be real-time.	A news site homepage that updates every hour.

5. How do you create a Next.js application?

The easiest way is to use `create-next-app`, which sets up a new Next.js project with all the necessary configurations.

```
Bash

# Using npm
npx create-next-app@latest my-next-app

# Using yarn
yarn create next-app my-next-app
```

This command will prompt you to choose between TypeScript/JavaScript, ESLint, Tailwind CSS, etc., and will scaffold the basic project structure.

6. What is the role of the `pages/` directory?

In Next.js, the `pages/` directory is central to its file-based routing system. **Each file inside `pages/` becomes a route.**

- `pages/index.js` becomes the homepage (`/`).
- `pages/about.js` becomes the `/about` route.
- `pages/posts/first-post.js` becomes `/posts/first-post`.

It’s where you define your **UI components (pages)** and **API endpoints**.

7. What is the difference between `pages` and `components` in Next.js?

- **Pages**: These are React components that live inside the `pages/` directory and are associated with a specific route. They represent a single page of your application. They are the entry points for your application's different views.
- **Components**: These are reusable React components that do not have a direct route. They are typically stored in a separate `components/` directory (or similar, it's not a special folder). You use them to build the UI of your pages, such as a navigation bar, a card, or a form.

Example:

- `pages/blog.js` might be the page component.
- `components/Card.js` might be a reusable component used within the `blog` page.

8. How does file-based routing work in Next.js?

File-based routing is Next.js's default routing mechanism. It simplifies routing by mapping files in the `pages/` directory to URL paths.

- A file named `pages/about.js` automatically creates the route `/about`.
- A nested file `pages/blog/post-1.js` creates the route `/blog/post-1`.
- Dynamic routes are created using brackets, e.g., `pages/posts/[id].js` for `/posts/123`.

This convention-over-configuration approach makes routing intuitive and easy to manage.

9. What is the `public/` folder in Next.js?

The `public/` folder is used for static assets like **images, fonts, and favicons**.

- The files inside it are served directly from the root of the application.
- For example, an image at `public/images/logo.png` can be accessed in your code or browser via the path `/images/logo.png`.
- This is the best place to store assets that don't need to be processed by Webpack or other build tools.

10. What is `next.config.js` used for?

The next.config.js file is an optional file in the root of your project that allows you to configure Next.js. It's where you can override the default settings.

Common uses include:

- Configuring **custom headers**.
- Setting up **rewrites** and **redirects**.
- Managing **environment variables**.
- Optimizing **images**.
- Customizing build settings.

Example:

JavaScript

```
// next.config.js
module.exports = {
  // Add an image domain for Next.js Image component
  images: {
    domains: ['example.com'],
  },
  // Add a redirect
  async redirects() {
    return [
      {
        source: '/old-about',
        destination: '/about',
        permanent: true,
      },
    ];
  },
};
```

---

2. Routing & Navigation

11. How to create dynamic routes in Next.js?

Dynamic routes are used when you don't know the exact names of your pages at build time, such as a product page for a product with an ID, or a blog post page.

You create them by using **brackets [] in the filename**. For example, to create a route for a blog post, you'd create a file `pages/posts/[slug].js`. The `[slug]` part becomes a dynamic parameter.

In the component, you can access this parameter using the `useRouter` hook from `next/router` (for the `pages` directory):

JavaScript

```
// pages/posts/[slug].js
import { useRouter } from 'next/router';

function Post() {
  const router = useRouter();
  const { slug } = router.query;

  return <h1>Post: {slug}</h1>;
}

export default Post;
```

12. What is the difference between `[id].js`, `[...slug].js`, and `[[...slug]].js`?

- `[id].js`: This is a **single dynamic segment**. It matches a single path segment.
    - **Matches**: `/posts/123`, `/posts/abc`
    - **Doesn't Match**: `/posts/`, `/posts/123/456`
- `[...slug].js`: This is a **catch-all dynamic segment**. It matches one or more path segments. The value is an array of segments.
    - **Matches**: `/posts/a`, `/posts/a/b`, `/posts/a/b/c`
    - **Doesn't Match**: `/posts/` (the last segment must be present)
- `[[...slug]].js`: This is an **optional catch-all dynamic segment**. It matches zero, one, or more path segments. This is useful for creating a single page that serves as both the index and the dynamic sub-pages.
    - **Matches**: `/posts/`, `/posts/a`, `/posts/a/b/c`

13. How to use `next/link` for navigation?

The next/link component is Next.js's primary way to navigate between pages. It's a simple wrapper around an <a> tag that handles client-side navigation and prefetching.

Key benefits:

- **Client-side navigation**: No full page reload.
- **Automatic prefetching**: Next.js automatically prefetches the code for linked pages when they appear in the viewport, making navigation instant.
- **SEO-friendly**: It still renders an `<a>` tag, so search engines can crawl it.

Example:

```
JavaScript

import Link from 'next/link';

function Navigation() {
  return (
    <nav>
      <Link href="/">
        <a>Home</a>
      </Link>
      <Link href="/about">
        <a>About</a>
      </Link>
    </nav>
  );
}
```

Note: Starting with Next.js 13, the `<a>` tag is not needed inside `Link`. You can directly use `<Link href="/about">About</Link>`.

14. Difference between client-side navigation and server-side navigation.

- **Client-side navigation**: This is the default behavior of `next/link`. When you click a link, Next.js intercepts the request, fetches the new page's JavaScript and data, and then updates the DOM **without a full page reload**. This results in a fast, SPA-like experience.
- **Server-side navigation**: This occurs when you use a regular `<a>` tag or when a route transition is forced. The browser sends a new request to the server, which then renders a completely new page and sends it back. This is a full page reload and is generally slower.

15. What is shallow routing in Next.js?

Shallow routing allows you to **change the URL without re-running data-fetching methods** like `getServerSideProps`, `getStaticProps`, or `getInitialProps`. This is useful for updating query parameters, like a filter, sort, or search query, without re-rendering the entire page.

You enable it by passing `shallow: true` to `router.push` or `router.replace`.

Example:

```
JavaScript

import { useRouter } from 'next/router';

function ProductList() {
  const router = useRouter();

  const handleFilterChange = (filter) => {
    router.push(`/?filter=${filter}`, undefined, { shallow: true });
  };

  // ...
}
```

16. How to implement nested routes in Next.js?

Nested routes are created by nesting folders within the `pages/` directory. For example, to create a blog with posts, you would have:

- `pages/blog/index.js` (for the blog homepage)
- `pages/blog/[slug].js` (for individual blog posts)

This structure automatically creates the `/blog` and `/blog/[slug]` routes.

17. How to handle query parameters in Next.js routes?

Query parameters are handled by the `router.query` object from the `useRouter` hook. It's an object where keys are the query parameter names and values are their respective values.

Example: For the URL `/posts/123?author=john`, `router.query` would be `{ id: '123', author: 'john' }`.

```
JavaScript

// pages/posts/[id].js
import { useRouter } from 'next/router';

function Post() {
  const router = useRouter();
  const { id, author } = router.query;

  return (
    <>
      <h1>Post ID: {id}</h1>
      <p>Author: {author}</p>
    </>
  );
}
```

18. Difference between static and dynamic imports in Next.js.

- **Static Imports**: This is the standard ES6 import (`import MyComponent from './MyComponent'`). The code is **bundled with the main JavaScript bundle** at build time. This is the default and is suitable for components that are used on every page.
- **Dynamic Imports**: This allows you to load JavaScript modules **on-demand** at runtime. Next.js has a built-in `next/dynamic` utility for this. It's used for **code splitting**, meaning the code for the dynamically imported component is in its own separate chunk, which is only loaded when needed. This improves initial page load performance.

Example:

```
JavaScript

import dynamic from 'next/dynamic';

const DynamicComponent = dynamic(() => import('../components/DynamicComponent'));

function MyPage() {
  return <DynamicComponent />;
}
```

19. How to add custom 404 and 500 pages in Next.js?

- **404 Page (Not Found)**: To create a custom 404 page, simply create a file named `pages/404.js`. Next.js will automatically serve this page for any route that doesn't exist.

- **500 Page (Server Error)**: To create a custom 500 page, create a file named `pages/500.js`. This page will be rendered if an error occurs on the server side (e.g., in `getServerSideProps`).

You can customize these pages with your own design and messaging.

20. What is middleware in Next.js (introduced in v12+)?

Next.js **Middleware** is a new feature that lets you run code **before a request is completed**. It's similar to Express middleware but runs on the edge, which means it's extremely fast. You can use it to:

- **Rewrite** or **redirect** a request.
- **Add HTTP headers** based on the request.
- **Authenticate** or authorize a user before they can access a page.
- Perform A/B testing.

To use it, create a `middleware.js` or `_middleware.js` file in the root of your project or within a subdirectory.

Example:

```
JavaScript

// middleware.js
import { NextResponse } from 'next/server';

export function middleware(request) {
  const token = request.cookies.get('token');

  // If the user is not authenticated, redirect them to the login page
  if (!token && request.nextUrl.pathname.startsWith('/dashboard')) {
    return NextResponse.redirect(new URL('/login', request.url));
  }

  return NextResponse.next();
}
```

---

3. Data Fetching

21. What is `getStaticProps()` in Next.js?

`getStaticProps` is a Next.js data-fetching function that runs **only on the server at build time**. It's used to fetch data for a page that will be **statically generated**.

- It's **not available on the client-side**.
- The data it fetches is passed to the page component as `props`.
- The result is a pre-rendered HTML page with the data already in place, making it extremely fast.

Example:

```
JavaScript

// pages/blog/[slug].js
export async function getStaticProps(context) {
  const { slug } = context.params;
  const post = await getPostBySlug(slug); // Fetch data from a CMS or file

  return {
    props: {
      post,
    },
  };
}

function PostPage({ post }) {
  return (
    <article>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </article>
  );
}
```

22. What is `getServerSideProps()` in Next.js?

`getServerSideProps` is a data-fetching function that runs **on the server for every request**. It is used for pages that require **frequently updated data** that must be fresh with every visit.

- It's an ideal choice for pages that need to be server-rendered.
- The data is passed to the page component as `props`.
- Since it runs on every request, it's slower than `getStaticProps` but ensures the data is always up-to-date.

Example:

```
JavaScript

// pages/products/[id].js
export async function getServerSideProps(context) {
  const { id } = context.params;
  const res = await fetch(`https://api.example.com/products/${id}`);
  const product = await res.json();

  return {
    props: {
      product,
    },
```

```
  };
}

function ProductPage({ product }) {
  return <h1>{product.name}</h1>;
}
```

23. Difference between `getStaticProps` and `getServerSideProps`.

Feature	getStaticProps	getServerSideProps
Execution	**Build time** (server-side)	**Per request** (server-side)
Data Freshness	Data is static until re-built or revalidated (ISR)	Data is always fresh and real-time
Use Case	Static content (blogs, documentation, marketing pages)	Dynamic, user-specific, or frequently changing data
Performance	**Extremely fast**, served from a CDN	Slower than SSG as it computes on every request
SEO	Excellent	Excellent

24. What is `getStaticPaths()`? Why is it used?

`getStaticPaths` is a function used with **dynamic routes** and `getStaticProps` to specify which paths should be **statically generated at build time**. It's essential for a site that uses SSG for dynamic content.

Why is it used? Because Next.js doesn't know what values `[slug]` can take, you have to tell it which paths to pre-render.

Example:

```
JavaScript

// pages/posts/[slug].js
```

```
export async function getStaticPaths() {
  const posts = await getAllPosts(); // Fetches a list of all posts
  const paths = posts.map(post => ({
    params: { slug: post.slug },
  }));

  return {
    paths, // [{ params: { slug: 'my-first-post' } }, { params: { slug: 'another-post' } }]
    fallback: false, // or 'blocking' or true
  };
}

export async function getStaticProps(context) {
  const { slug } = context.params;
  const post = await getPostBySlug(slug);
  return { props: { post } };
}
```

25. Difference between `getInitialProps()` and `getServerSideProps()`.

`getInitialProps` was the original data-fetching method in Next.js. It's a legacy function that runs on both the **server and the client** (during client-side navigation). This dual-execution model can be confusing and lead to performance issues.

`getServerSideProps` is the modern, recommended alternative. It **only runs on the server**, which makes the logic clearer and prevents code from being sent to the client. This is a significant improvement. You should almost always prefer `getServerSideProps` over `getInitialProps`.

26. What is Incremental Static Regeneration (ISR)?

ISR is a feature that allows you to **update statically generated pages without a full site rebuild**. It's a compromise between the speed of SSG and the freshness of SSR.

How it works:

- You use `getStaticProps` with a `revalidate` property.
- Next.js generates the page at build time.
- When a user requests the page after the `revalidate` time has passed, they are served the **stale (cached) page**.
- In the background, Next.js regenerates the page with the new data.
- The next user to request the page will see the newly regenerated page.

27. How does revalidation work in ISR?

Revalidation in ISR is handled by the `revalidate` key in the `getStaticProps` return object.

Example:

```
JavaScript
```

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/data');
  const data = await res.json();

  return {
    props: { data },
    revalidate: 60, // Revalidate this page every 60 seconds
  };
}
```

- The page is first built and served.
- A user visits the page. The content is served instantly.
- Another user visits at `T + 59s`. They also get the same cached content.
- A user visits at `T + 61s`. They still get the old content, but Next.js **triggers a background regeneration** of the page.
- A user visits at `T + 62s`. The new content is now ready, so they get the fresh version.

This ensures that users always get a fast page load while the content is updated in the background.

28. Difference between static generation and server-side rendering in terms of SEO.

Both SSG and SSR are **excellent for SEO** because they deliver a fully rendered HTML page to the browser. Search engine crawlers (like Googlebot) can easily read the content, index it, and understand its structure.

- **SSG**: SEO is top-notch. The content is baked into the HTML at build time, and the page is served from a CDN with a super-fast load time, which is a key ranking factor for Google.
- **SSR**: SEO is also very good. The page is rendered on the server, so crawlers see the full content. While it's slightly slower than SSG, it's still much better for SEO than CSR, where crawlers would have to execute JavaScript to see the content.

29. When should you use client-side data fetching (`useEffect`) instead of SSR/SSG?

You should use client-side data fetching with useEffect for user-specific, dynamic data that doesn't need to be indexed by search engines.

Ideal use cases:

- A **dashboard** for a logged-in user.
- Data that's only visible **after a user action**, like a form submission.
- Content that is not critical for the initial page load (e.g., a "related products" section).

For this approach, you would render a loading state on the server (e.g., `Loading...`) and then use a `useEffect` hook to fetch the data after the component mounts on the client.

30. How to fetch data from APIs inside Next.js pages?

You can fetch data inside Next.js pages in several ways:

1. **Server-side (for SSR/SSG)**: Use `getServerSideProps` or `getStaticProps`. This is the recommended approach for SEO-critical data.
2. **Client-side**: Use the `useEffect` hook with a library like **SWR** or **React Query**. These libraries are great because they handle caching, re-fetching, and state management for you.

Example with `useEffect` and `fetch`:

```
JavaScript

import { useEffect, useState } from 'react';

function MyPage() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    async function fetchData() {
      const res = await fetch('/api/data');
      const data = await res.json();
      setData(data);
      setLoading(false);
    }
    fetchData();
  }, []);

  if (loading) return <p>Loading...</p>;

  return <div>{data.message}</div>;
}
```

---

4. API Routes

31. What are API routes in Next.js?

API Routes are a feature of Next.js that allows you to create a backend API within your frontend application. Any file inside the `pages/api/` directory becomes an API endpoint. This means you can build a full-stack application with a single codebase.

They provide a serverless function-like environment, perfect for tasks like:

- Connecting to a database.
- Handling form submissions.
- Managing user authentication.

32. How to create an API endpoint in Next.js?

You create an API endpoint by creating a file inside the `pages/api/` directory. The file must export a **default function** that receives a `request` object and a `response` object.

Example: `pages/api/hello.js`

```
JavaScript

export default function handler(req, res) {
  res.status(200).json({ message: 'Hello from the API!' });
}
```

This creates an endpoint at `/api/hello`.

33. Difference between Next.js API routes and Express.js routes.

Feature	Next.js API Routes	Express.js Routes
Paradigm	**File-based** and serverless function-like	**Explicitly defined** with `app.get()`, `app.post()`, etc.
Setup	**Built-in** with Next.js; no extra setup required	Requires a separate project, and a dedicated server (e.g., `index.js`)
Deployment	Deployed as **serverless functions** on platforms like Vercel	Deployed on a server that is always running (e.g., EC2, Heroku)
Use Case	Ideal for lightweight backend logic within a Next.js app	Full-fledged backends with complex business logic, many endpoints, and middleware

34. How to handle POST requests in Next.js API routes?

The `request` object has a `method` property that tells you the HTTP method of the incoming request. You can use a conditional statement to handle different methods.

Example: `pages/api/submit-form.js`

```
JavaScript
```

```
export default function handler(req, res) {
  if (req.method === 'POST') {
    // Process a POST request
    const { name } = req.body;
    res.status(200).json({ message: `Thanks, ${name}!` });
  } else {
    // Handle any other HTTP method
    res.setHeader('Allow', ['POST']);
    res.status(405).end(`Method ${req.method} Not Allowed`);
  }
}
```

35. How to connect Next.js API routes to a database (MongoDB, PostgreSQL)?

To connect to a database from an API route, you typically use a database driver or an ORM library.

1. **Install the library**: e.g., `npm install mongodb` or `npm install pg`.
2. **Create a utility function**: It's a best practice to create a separate file (e.g., `lib/db.js`) that handles the database connection.
3. **Call the function in your API route**: Import the connection utility and use it to perform database operations.

Example with MongoDB:

JavaScript

```
// lib/db.js
import { MongoClient } from 'mongodb';

let client;
export async function connectToDatabase() {
  if (!client) {
    client = await MongoClient.connect(process.env.MONGODB_URI);
  }
  return client;
}

// pages/api/users.js
import { connectToDatabase } from '../../lib/db';

export default async function handler(req, res) {
  const client = await connectToDatabase();
  const db = client.db('my-database');
  const users = await db.collection('users').find({}).toArray();
  res.status(200).json({ users });
}
```

36. What is the best way to secure API routes in Next.js?

Securing API routes involves several key practices:

1. **Authentication**: Use a library like **NextAuth.js** to handle user sign-in and protect routes with a session or JWT token.
2. **Authorization**: Check if the authenticated user has the necessary permissions to perform an action.
3. **Environment Variables**: Never hardcode secrets like API keys, database credentials, or private keys. Use environment variables (`.env.local`).
4. **CORS**: Only allow requests from trusted origins. By default, Next.js API routes are CORS-restricted.
5. **Input Validation**: Sanitize and validate all incoming data to prevent security vulnerabilities like SQL injection or cross-site scripting (XSS).
6. **Rate Limiting**: Limit the number of requests a single IP can make to prevent abuse.

37. How to use middleware with API routes?

Next.js API routes don't have built-in middleware support like Express.js. To implement middleware-like behavior, you have a few options:

1. **Manual Check**: Write conditional logic inside each handler function to check for things like authentication. This is fine for simple cases but can lead to code duplication.
2. **Higher-Order Function (HOF)**: Create a reusable function that wraps your API route handler.

Example with HOF:

JavaScript

```
// middleware/withAuth.js
export const withAuth = (handler) => async (req, res) => {
  const token = req.headers.authorization;
  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }
  // Check token and attach user to req object
  // ...
  return handler(req, res);
};

// pages/api/protected-route.js
import { withAuth } from '../../middleware/withAuth';

const handler = (req, res) => {
  res.status(200).json({ message: 'This is a protected route' });
};

export default withAuth(handler);
```

38. Difference between pages API routes vs. custom server in Next.js.

- **API Routes (built-in)**: This is the **standard and recommended approach**. It's serverless by design, meaning it's highly scalable and cost-effective on platforms like Vercel. You don't need to manage a server, and each API route is treated as a separate function.
- **Custom Server**: A custom server (e.g., using Express.js) is an alternative to API routes. It gives you more control over the server, allows you to use your favorite middleware, and integrates with existing backend code. However, it **disables serverless features** and requires you to manage the server yourself. This is typically used for complex, specific needs.

39. Can Next.js replace Express.js as a backend?

Yes, for many use cases, Next.js API routes can replace Express.js.

- For **full-stack applications** where the backend logic is tightly coupled with the frontend (e.g., fetching data, handling form submissions), API routes are a perfect fit. They simplify the project structure and deployment.
- However, for **complex, microservices-based architectures** or applications that require an independent, highly custom backend (e.g., a GraphQL server, a separate payment gateway), an independent Express.js or similar backend is still the better choice.

40. What are edge API routes in Next.js 13?

Edge API routes, or **Edge Functions**, are a new type of API route that runs on a distributed network of servers called the **Edge**. Unlike standard Node.js serverless functions, Edge Functions are extremely lightweight and are executed **physically closer to the user**.

- They are ideal for tasks that need to be executed with **minimal latency**, such as A/B testing, authentication, or redirects.
- They have a **smaller API surface** and limited access to Node.js APIs, so they are not a full replacement for Node.js serverless functions.
- You can create them using the new App Router and the Edge Runtime.

---

5. Styling in Next.js

41. What styling options are available in Next.js?

Next.js is unopinionated about styling and supports a wide range of popular solutions out-of-the-box:

1. **CSS Modules**: The recommended way to style components locally.
2. **Sass/SCSS**: Provides support for preprocessors.
3. **Styled JSX**: A CSS-in-JS solution built-in by Vercel.
4. **Tailwind CSS**: A utility-first CSS framework.
5. **Styled Components / Emotion**: Popular CSS-in-JS libraries.

42. How to use CSS Modules in Next.js?

CSS Modules scope your CSS classes to a specific component, preventing naming conflicts.

1. Create a CSS file with the `.module.css` extension, e.g., `styles/Home.module.css`.
2. Import the file into your component.
3. Use the imported object to apply classes.

Example:

CSS

```
/* styles/Home.module.css */
.title {
  color: #0070f3;
  font-size: 2rem;
}
```

JavaScript

```
// pages/index.js
import styles from '../styles/Home.module.css';

function HomePage() {
  return <h1 className={styles.title}>Welcome!</h1>;
}
```

43. Difference between global CSS and module CSS.

- **Global CSS**: CSS files imported into `pages/_app.js` are treated as global styles. They affect every single component and page in your application. Use this for things like a reset, base typography, or global layout styles.
- **Module CSS**: CSS files with `.module.css` are scoped locally to the component they are imported into. The class names are hashed at build time, preventing conflicts. This is the **recommended method** for component-level styling.

44. What is Styled JSX in Next.js?

Styled JSX is a **CSS-in-JS library** from Vercel that is built into Next.js. It allows you to write CSS directly inside your components within a `<style jsx>` tag. The styles are scoped to the component, so you don't have to worry about conflicts.

Example:

JavaScript

```
function Button() {
  return (
    <>
```

```
      <button>Click Me</button>
      <style jsx>{`
        button {
          padding: 1rem 2rem;
          background-color: #0070f3;
          color: white;
        }
      `}</style>
    </>
  );
}
```

45. How to use Sass/SCSS in Next.js?

To use Sass, you first need to install the `sass` package:

```
Bash

npm install sass
```

Then, you can create files with a .scss or .sass extension and import them as you would with regular CSS files. Next.js automatically handles the compilation.

You can use it both globally (_app.scss) and as modules ([name].module.scss).

46. How to configure Tailwind CSS in Next.js?

1. **Install dependencies**: `npm install -D tailwindcss postcss autoprefixer`.
2. **Initialize Tailwind**: `npx tailwindcss init -p`. This creates `tailwind.config.js` and `postcss.config.js`.
3. **Configure `tailwind.config.js`**: Add the paths to all your component files so Tailwind can scan them for classes.
4. JavaScript

```
// tailwind.config.js
module.exports = {
  content: [
    './pages/**/*.{js,ts,jsx,tsx}',
    './components/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

5.
6.

7. **Import base styles**: Create a styles/globals.css file and add the Tailwind directives.
8. CSS

```
/* styles/globals.css */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

9.
10.
11. **Use classes**: Now you can use Tailwind classes directly in your JSX.

47. What is CSS-in-JS? Examples in Next.js.

CSS-in-JS is a pattern where you write CSS code directly within your JavaScript files. It's popular for creating scoped styles, handling dynamic styles easily, and providing better component encapsulation.

Examples in Next.js:

- **Styled JSX**: The built-in solution.
- **Styled Components**: A very popular library. You install it, configure the babel-plugin-styled-components in next.config.js, and then use its API to create styled components.
- **Emotion**: Another popular library similar to Styled Components.

48. How to use dynamic class names in Next.js?

You can use template literals with JavaScript to create dynamic class names.

- For **CSS Modules**, you can combine static and dynamic classes.
- JavaScript

```
import styles from './styles.module.css';

const isActive = true;
const buttonClass = `${styles.button} ${isActive ? styles.active : ''}`;

<button className={buttonClass}>Click</button>
```

-
-
- For **Tailwind**, you can use a library like clsx or classnames to conditionally join class strings.
- JavaScript

```
import clsx from 'clsx';
```

```
const isActive = true;
const buttonClass = clsx('p-4 rounded', {
  'bg-blue-500 text-white': isActive,
  'bg-gray-300': !isActive,
});

<button className={buttonClass}>Click</button>
```

-
-

49. Difference between inline styles and CSS Modules.

- **Inline Styles**: CSS properties are written directly as a JavaScript object on an element's `style` attribute (`<div style={{ color: 'red' }}>...</div>`). This is great for simple, dynamic styles but has several downsides: no pseudo-selectors (`:hover`), no media queries, and they can make your markup cluttered.
- **CSS Modules**: You write your CSS in a separate file, which is then imported and applied with `className`. This allows you to use the full power of CSS, including pseudo-selectors, media queries, and animations, while still having scoped styles.

Recommendation: Use **CSS Modules** for most styling. Use inline styles only for truly dynamic, simple properties.

50. Best practices for styling in large-scale Next.js projects.

1. **Start with CSS Modules**: They offer the perfect balance of encapsulation, maintainability, and performance for most components.
2. **Use Global CSS sparingly**: Reserve `globals.css` for things like CSS resets, fonts, and shared variables.
3. **Choose a consistent naming convention**: Stick to a system like BEM or a utility-first approach like Tailwind.
4. **Use a preprocessor (Sass)** if you need nesting, variables, or mixins.
5. **Document your styling choices**: A large project can quickly become difficult to manage without clear guidelines.
6. **Consider a UI library**: For complex components, using a pre-built and well-documented UI library (e.g., Material-UI, Ant Design) can save significant development time.

---

6. Performance Optimization

51. How does Next.js improve SEO?

Next.js improves SEO by providing multiple **rendering strategies that deliver fully-formed HTML** to the browser.

- **Server-Side Rendering (SSR)**: Generates HTML on the server, which is easily readable by search engine crawlers.

- **Static Site Generation (SSG)**: Pre-renders pages at build time. The resulting static files are incredibly fast and can be served from a CDN, which is a major factor in Google's ranking algorithm.
- **Automatic Code Splitting**: Next.js automatically splits your JavaScript bundles per page, so a user only downloads the code needed for that specific page, leading to faster load times.
- **Image Optimization**: The `next/image` component optimizes images automatically.
- **Built-in Metadata Management**: The `next/head` component allows you to easily manage `<head>` tags for each page, including title, meta descriptions, and open graph tags.

52. What is Automatic Image Optimization in Next.js (`next/image`)?

The next/image component is a built-in feature that provides automatic image optimization. It's a key performance tool.

It automatically handles:

1. **Resizing**: Generates multiple image sizes and serves the optimal size for each device.
2. **Format Conversion**: Converts images to modern formats like **WebP** if the browser supports them.
3. **Lazy Loading**: Images that are not in the viewport are lazy-loaded by default.
4. **Blur-up Placeholder**: Provides a low-quality placeholder until the high-quality image loads.
5. **Preventing Layout Shift**: It automatically calculates the `width` and `height` to prevent layout shift.

Example:

```
JavaScript

import Image from 'next/image';
import myImage from '../public/my-image.jpg';

function MyComponent() {
  return (
    <Image
      src={myImage}
      alt="A description of my image"
      width={500}
      height={300}
    />
  );
}
```

53. How does code splitting work in Next.js?

Code splitting is the process of splitting a single large JavaScript bundle into multiple smaller chunks. Next.js does this automatically on a **per-page basis**.

- When a user visits a page, they only download the JavaScript code required for that page, not the entire application.
- Next.js also handles the splitting of shared code (e.g., React library, reusable components) into a separate chunk that is cached by the browser.
- This significantly reduces the initial page load time, especially on large applications.

54. What is lazy loading in Next.js?

Lazy loading is a technique that defers the loading of non-critical resources (like images, components, or modules) until they are needed.

- **next/image**: As mentioned, `next/image` lazy loads images by default.
- **next/dynamic**: You can lazy load components using `next/dynamic`. The component's code is not loaded until the component is rendered. This is perfect for components that are only visible after a user action, like a modal or a comment section.

55. How to optimize font loading in Next.js?

Next.js provides a built-in `next/font` module that handles font optimization automatically. It:

1. **Eliminates external network requests**: Fonts are self-hosted and loaded at build time.
2. **Prevents layout shift (CLS)**: It handles font loading to prevent the "Flash of Unstyled Text" (FOUT) and layout shifts.
3. **Handles performance**: It automatically sets up things like font preloading and caching.

Example:

JavaScript

```
// pages/_app.js
import { Inter } from 'next/font/google';

const inter = Inter({ subsets: ['latin'] });

function MyApp({ Component, pageProps }) {
  return (
    <main className={inter.className}>
      <Component {...pageProps} />
    </main>
  );
}
```

56. What is static optimization in Next.js?

Static optimization is a core performance feature of Next.js. It's the process by which Next.js can identify pages that can be **statically generated** without needing a data-fetching method (`getStaticProps` or `getServerSideProps`).

- For a simple page that doesn't have any data-fetching functions, Next.js will automatically pre-render it as a static HTML file at build time.
- This makes these pages lightning-fast and deployable to a CDN, similar to a static site.

57. How does Next.js handle prefetching of routes?

`next/link` handles prefetching automatically. When a `Link` component enters the viewport, Next.js will **prefetch the JavaScript code for the linked page in the background**.

- When a user then clicks the link, the page transition is instant because the necessary code has already been downloaded.
- This is a key reason why client-side navigation feels so fast in Next.js.
- You can disable prefetching by setting the `prefetch` prop to `false`.

58. How to optimize large Next.js applications?

1. **Prioritize SSG/ISR**: Use static generation and incremental static regeneration for as many pages as possible to leverage CDN caching.
2. **Use `next/image` and `next/font`**: Utilize Next.js's built-in optimization for images and fonts.
3. **Implement dynamic imports**: Lazily load components that are not critical for the initial view.
4. **Use a data-fetching library (SWR/React Query)**: These libraries handle caching, re-fetching, and state management, reducing unnecessary network requests.
5. **Use the `next/bundle-analyzer`**: This tool helps you visualize the size of your JavaScript bundles so you can identify and optimize large dependencies.
6. **Optimize your API routes**: Ensure they are fast, efficient, and only return the necessary data.

59. What is edge rendering in Next.js 13?

Edge Rendering is a feature of the new App Router that allows you to render pages at the edge, closer to the user. It leverages the new **React Server Components** and the **Edge Runtime** to achieve this.

- It's different from both SSR (rendering on a central server) and SSG (rendering at build time).
- Edge rendering is fast and stateless, making it ideal for tasks that require a quick, dynamic response but don't need a heavy-duty Node.js server.
- You can opt-in to edge rendering by specifying the `runtime` in your page or API route.

60. Difference between caching in ISR and SSR.

- **ISR Caching**: Pages are cached on a **CDN**. The HTML is generated at build time, and subsequent requests are served from the cache. The cache is only invalidated

when the `revalidate` time passes or a revalidation request is made. This is highly efficient and scalable.
- **SSR Caching**: Pages are **not cached by Next.js** by default. The server generates a new HTML file for **every request**. You can implement your own caching mechanisms on the server (e.g., using a reverse proxy like Nginx or a cache-control header), but it is not handled by Next.js itself.

---

7. Authentication & Security

61. How to implement authentication in Next.js?

Implementing authentication in Next.js typically involves a few key steps:

1. **Choose a strategy**: Decide between JWT (stateless) and session-based (stateful) authentication. JWT is common with Next.js because it's stateless, making it easy to scale.
2. **Create API routes**: Create API endpoints for login, logout, and user registration.
3. **Protect routes**: Use a middleware or a check inside `getServerSideProps` to ensure a user is authenticated before they can access a protected page.
4. **Manage state**: Use React context or a global state management library to keep track of the user's authentication status on the client.
5. **Store tokens**: Use cookies to securely store JWTs or session tokens.

62. What is NextAuth.js? How to use it?

NextAuth.js is a complete open-source authentication library for Next.js. It simplifies authentication by providing a pre-built solution that handles:

- **Sign-in with various providers**: Google, GitHub, Auth0, etc.
- **Email/passwordless authentication**.
- **Session management**: It uses JWT by default and is highly secure.
- **Protecting API routes and pages**.

To use it:

1. Install the package: `npm install next-auth`.
2. Create `pages/api/auth/[...nextauth].js` and configure your providers.
3. Wrap your `_app.js` with the `SessionProvider`.
4. Use the `useSession` hook on the client to check the user's status.

63. Difference between session-based and JWT authentication in Next.js.

Feature	Session-Based Authentication	JWT (JSON Web Token) Authentication

Mechanism	Server creates a session and stores a unique ID (in a cookie). The server maps the ID to user data.	Server creates a token that contains user data. The token is sent to the client and stored.
State	**Stateful**: The server must maintain the session state (e.g., in memory, database, Redis).	**Stateless**: The server doesn't need to store session information. It verifies the token on each request.
Scalability	Can be challenging for scaling (requires a shared session store).	Highly scalable and ideal for distributed or serverless architectures.
Use Case	Traditional web applications with a single backend.	SPAs, mobile apps, and microservices-based architectures.

64. How to protect API routes with authentication?

You can protect API routes by checking for a valid authentication token in the request headers or cookies.

- **Manual check**: Use the higher-order function pattern mentioned in a previous answer.
- **NextAuth.js**: Use the `getSession` function to check for an active session in the API route. If there's no session, return a 401 Unauthorized response.

Example with NextAuth.js:

JavaScript

```
// pages/api/protected.js
import { getSession } from 'next-auth/react';

export default async function handler(req, res) {
  const session = await getSession({ req });

  if (!session) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  // User is authenticated, proceed with the API logic
```

```
  res.status(200).json({ message: 'This is a protected resource' });
}
```

65. How to use middleware for route protection?

Next.js middleware (v12+) is a great way to protect entire routes and directories.

1. Create a `middleware.js` file in the root of your project.
2. Inside the middleware function, check for a user's token or session.
3. If the user is not authenticated and is trying to access a protected route, you can redirect them to a login page.

Example:

```
JavaScript

// middleware.js
import { NextResponse } from 'next/server';

export function middleware(request) {
  const token = request.cookies.get('token');
  const pathname = request.nextUrl.pathname;

  if (!token && pathname.startsWith('/dashboard')) {
    return NextResponse.redirect(new URL('/login', request.url));
  }

  return NextResponse.next();
}
```

66. How to store tokens securely in Next.js (cookies vs localStorage)?

Use cookies, not `localStorage` or `sessionStorage`, for storing tokens.

- **Cookies**: They are sent with every HTTP request automatically, which is convenient for a full-stack framework. You can also set the `httpOnly` flag on cookies, which makes them inaccessible to client-side JavaScript. This is the **best way to prevent XSS attacks** from stealing the token.
- **`localStorage`/`sessionStorage`**: They are vulnerable to XSS attacks. If an attacker can inject malicious JavaScript into your site, they can easily access and steal the user's token, which can then be used to impersonate the user.

67. What is CSRF and how to prevent it in Next.js?

Cross-Site Request Forgery (CSRF) is an attack where an attacker tricks a user's browser into sending a malicious request to a web application they are authenticated with.

Prevention:

1. **Same-Site Cookies**: The `SameSite` attribute on cookies can prevent the browser from sending a cookie with cross-site requests. This is a very effective and modern defense.
2. **CSRF Tokens**: The server generates a unique, random token for each user session. The token is sent to the client and included with every form submission or API request. The server then validates the token. If the tokens don't match, the request is rejected.

You can implement this manually in your API routes or use a library that handles it for you.

68. How to implement role-based authentication in Next.js?

Role-based authentication (RBAC) involves checking a user's role (e.g., `admin`, `editor`, `user`) to determine what actions they are allowed to perform.

1. **Store roles**: The user's role should be included in their session or JWT payload.
2. **Protect pages**: Use `getServerSideProps` to check the user's role. If they don't have the correct role, redirect them or show an error.
3. **Protect API routes**: Inside your API handler, check the user's role before performing any action.

Example:

```
JavaScript

// pages/admin/dashboard.js
import { getSession } from 'next-auth/react';

export async function getServerSideProps(context) {
  const session = await getSession(context);

  if (!session || session.user.role !== 'admin') {
    return {
      redirect: {
        destination: '/',
        permanent: false,
      },
    };
  }

  return { props: { user: session.user } };
}
```

69. Difference between client-side and server-side authentication.

- **Client-side Authentication**: The token is checked on the client. The page is loaded, and then the client-side code checks if the user is authenticated. This approach is **insecure** and not recommended for protecting sensitive content because the page's source code is accessible to anyone.

- **Server-side Authentication**: The authentication check happens on the server before the page is rendered and sent to the client. This is the **secure and recommended approach**. It prevents unauthorized users from even seeing the page's source code. You can achieve this using `getServerSideProps` or Next.js middleware.

70. Best practices for securing a Next.js app.

1. **Use `getServerSideProps` or Middleware for protection**: Always perform authentication and authorization checks on the server side.
2. **Store tokens in `httpOnly` cookies**: Prevent client-side JavaScript from accessing and potentially stealing the user's session.
3. **Use a robust authentication library**: Libraries like NextAuth.js handle many security concerns for you, like CSRF, session management, and JWT signing.
4. **Validate all user input**: Sanitize and validate data on both the client and the server to prevent common attacks like XSS and SQL injection.
5. **Use environment variables**: Never hardcode sensitive information. Use `.env.local` for development and configure them in your hosting provider for production.
6. **Regularly update dependencies**: Stay on top of security patches by keeping your packages up-to-date.

---

8. Advanced Features

71. What is Next.js middleware (v12+)?

Middleware is a Next.js feature that allows you to run code **before a request is completed** and a page is rendered. It's a powerful tool for global logic that needs to happen for every request.

- **Location**: It lives in `middleware.js` at the root of your project or in a subdirectory.
- **Execution**: It runs on the Edge, which is a global network of servers, making it incredibly fast.
- **Use cases**: Rewriting URLs, redirecting users, adding HTTP headers, and authenticating users.

72. How to use Edge Functions in Next.js?

Edge Functions are a type of serverless function that run on the Edge Runtime. They are lightweight, stateless, and execute with very low latency.

- **How to create one**:
    - In the `pages` directory: an API route can opt-in to the Edge runtime by exporting a `config` object:
    - JavaScript

```
export const config = {
  runtime: 'edge',
```

};

```
```

```
```

    - In the new `app` directory: The `route.js` file can also be configured to run on the Edge.
- **Limitations**: They have a smaller API than Node.js, so you can't access things like the file system. They are best for simple logic like A/B testing or authentication.

73. What are React Server Components (Next.js 13)?

React Server Components (RSC) are a new paradigm introduced in Next.js 13's App Router. They are components that **render on the server only** and are never sent to the client.

- **Benefits**:
    - **Zero-client-side JavaScript**: They don't contribute to the client-side bundle, leading to smaller bundle sizes and faster load times.
    - **Direct access to server resources**: They can directly access databases, file systems, and other server-side resources without needing an API route.
    - **Improved SEO**: The server renders the initial HTML, providing excellent SEO.
- **How they work**: They are rendered on the server, and the result (which is not just HTML, but a special `RSC` payload) is streamed to the client, which then merges it with the client-side rendered UI.

74. Difference between `pages/` and new `app/` directory in Next.js 13.

The new app/ directory is an evolution of Next.js that brings in the new React features like Server Components.

| Feature | pages/ Directory (Traditional) | app/ Directory (Next.js 13+) |

|---|---|---|

| Rendering | Relies on getServerSideProps or getStaticProps | Defaults to Server Components; supports client components |

| Routing | File-based routing (index.js becomes /) | Folder-based routing (/dashboard/page.js becomes /dashboard) |

| Data Fetching | getStaticProps, getServerSideProps, getInitialProps | Standard fetch API, Server Actions, route handlers |

| State | All components are Client Components, so state is managed client-side | Can use server components (stateless) and client components (stateful) |

| Middleware | middleware.js file at the root | The same middleware.js file but with more advanced features. |

| Layouts | Manual layout management with _app.js and _document.js | Automatic nested layouts via layout.js files |

75. What is server-side streaming in Next.js?

Server-side streaming is a technique that allows a server to send a partially rendered HTML document to the browser while the rest of the page is still being rendered.

- Next.js 13's App Router supports this out-of-the-box.
- When a request comes in, the server immediately sends the static parts of the page (like a header or footer) and a loading state for the dynamic parts.
- As the dynamic components finish rendering on the server (e.g., after a database query), their HTML is streamed to the browser, replacing the loading state.
- This significantly improves the perceived performance of the page.

76. How does Next.js handle internationalization (i18n)?

Next.js has built-in support for internationalization (i18n).

1. **Configuration**: You configure your locales and default locale in `next.config.js`.
2. **Routing**: Next.js automatically handles routing for different locales (`/en/about`, `/fr/about`).
3. **Locale detection**: It can detect the user's preferred locale from the browser's language headers.

Example in `next.config.js`:

```
JavaScript

module.exports = {
  i18n: {
    locales: ['en', 'fr', 'es'],
    defaultLocale: 'en',
  },
};
```

For content translation, you can use a library like `next-i18next`.

77. What is static HTML export in Next.js?

A **static HTML export** is a feature that allows you to export your Next.js application as a set of static HTML, CSS, and JavaScript files.

- **Command**: `next export`.
- **Use case**: For applications that only use `getStaticProps` and do not need a Node.js server to run. This is ideal for a simple blog, a portfolio site, or a marketing page.
- **Limitations**: It **doesn't support SSR**, API routes, or middleware.

78. How to use environment variables in Next.js?

Next.js provides a secure way to manage environment variables.

- **Private variables**: Store them in `.env.local`. These are **only available on the server-side** (e.g., in `getStaticProps` or API routes).
- **Public variables**: Prefix a variable with `NEXT_PUBLIC_` to make it accessible on both the **server and the client**. Use these for things like a public API key.

Example:

```
# .env.local
DB_URI=mongodb://...
NEXT_PUBLIC_API_KEY=123-abc-456
```

JavaScript

```
// A server-side page
export function getServerSideProps() {
  const dbUri = process.env.DB_URI; // Available here
}

// A client-side component
function MyComponent() {
  const apiKey = process.env.NEXT_PUBLIC_API_KEY; // Available here
}
```

79. What is fallback rendering in Next.js ISR?

When using `getStaticPaths`, the `fallback` key determines how Next.js handles paths that were **not generated at build time**.

- `fallback: false`: If a user requests a path not in `getStaticPaths`, they see a 404 page.
- `fallback: 'blocking'`: Next.js will **block the request** and render the page on the server. Once rendered, it will be cached and served statically for all future requests. This is good for a very large number of static pages that don't need to be all generated at once.
- `fallback: true`: Next.js will **serve a fallback version** of the page (usually a loading state). In the background, it will generate the static page, and then once it's ready, the browser will receive the new content. Subsequent requests will be served the static page.

80. What are parallel and intercepting routes in Next.js 13?

- **Parallel Routes**: Allow you to **simultaneously render one or more pages within the same layout**. They are ideal for dashboards or complex UIs. For example, you can have a main content area and a sidebar, both with their own routing. You create them by using a folder prefixed with `@`, like `@analytics` or `@team`.
- **Intercepting Routes**: Allow you to **"intercept" a route and render a different UI** while maintaining the original URL in the browser. They are commonly used for

modals. For example, clicking on a photo in a gallery (`/photos/1`) can "intercept" the route and open a modal with the photo while keeping the underlying `/gallery` page visible. You create them using special conventions like `(..)` for a single level up.

---

9. Testing & Debugging

81. How to test Next.js applications?

Testing a Next.js app involves a combination of:

- **Unit testing**: Testing individual components in isolation.
- **Integration testing**: Testing how components work together.
- **End-to-end (E2E) testing**: Simulating a user's journey through the application.

Common tools include **Jest**, **React Testing Library**, and **Cypress**.

82. Difference between unit, integration, and E2E testing in Next.js.

Test Type	Description	Best For	Tooling
Unit Testing	Tests a single function or component in isolation.	Logic, simple components, and utilities.	**Jest**
Integration Testing	Tests how two or more components or systems work together.	A form and an API route, a parent and a child component.	**React Testing Library** (with Jest)
E2E Testing	Simulates a user's flow through the entire app in a real browser.	User registration, checkout flow, complex interactions.	**Cypress**, **Playwright**

83. How to use Jest with Next.js?

Next.js supports Jest out of the box with the `@next/jest` preset.

1. **Install Jest**: `npm install -D jest jest-environment-jsdom @next/jest`.
2. **Configure `jest.config.js`**:
3. JavaScript

```
const nextJest = require('@next/jest');
const createJestConfig = nextJest({ dir: './' });
const customJestConfig = {
  testEnvironment: 'jest-environment-jsdom',
};
module.exports = createJestConfig(customJestConfig);
```

4.
5.
6. **Write tests**: Create files with `.test.js` or `.spec.js` extension.

84. How to test API routes in Next.js?

Testing API routes is crucial. You can use a combination of **Jest** and a library like `supertest`.

1. **Import the API handler**: Import your API route function directly into your test file.
2. **Simulate a request**: Use `supertest` to create a mock `request` and `response` object.
3. **Assert the response**: Check the `res` object's status code, headers, and JSON body.

Example:

```
JavaScript

// pages/api/hello.test.js
import handler from './hello';

describe('/api/hello', () => {
  it('should return a JSON message', async () => {
    const req = { method: 'GET' };
    const res = {
      status: jest.fn(() => res),
      json: jest.fn(),
    };
    await handler(req, res);
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith({ message: 'Hello from the API!' });
  });
});
```

85. What is React Testing Library and how to use it with Next.js?

React Testing Library (RTL) is a library for testing React components. It focuses on testing components from a user's perspective, encouraging better practices.

- **Principle**: "The more your tests resemble the way your software is used, the more confidence they can give you."
- **Usage**: It's a key part of most Next.js testing setups. You use it to render components in a simulated browser environment and then query for elements on the screen.

Example:

JavaScript

```
// components/Button.test.js
import { render, screen } from '@testing-library/react';
import Button from './Button';

test('renders a button with the correct text', () => {
  render(<Button>Click Me</Button>);
  const buttonElement = screen.getByRole('button', { name: /click me/i });
  expect(buttonElement).toBeInTheDocument();
});
```

86. How to mock `getServerSideProps` and `getStaticProps` in tests?

You should **not** test these functions by trying to simulate the Next.js server. Instead, you should:

1. **Extract the logic**: Move the core data-fetching logic into a separate function that can be easily tested.
2. **Mock the data**: In your component test, mock the data returned by `getStaticProps` or `getServerSideProps`.

Example:

JavaScript

```
// lib/data.js
export async function getPosts() {
  const res = await fetch('...');
  return res.json();
}

// pages/blog.js
import { getPosts } from '../lib/data';

export async function getStaticProps() {
  const posts = await getPosts();
  return { props: { posts } };
}

// pages/blog.test.js
import { getStaticProps } from './blog';
import { getPosts } from '../lib/data';

// Mock the data fetching function
jest.mock('../lib/data');

test('getStaticProps should fetch posts', async () => {
  getPosts.mockResolvedValueOnce([{ title: 'Test Post' }]);
```

```
  const result = await getStaticProps();
  expect(result.props.posts).toEqual([{ title: 'Test Post' }]);
});
```

87. How to do end-to-end testing in Next.js with Cypress?

Cypress is a popular E2E testing framework. It runs tests in a real browser and can simulate user interactions.

1. **Install Cypress**: npm install -D cypress.
2. **Configure**: Add a cypress script to your package.json and a cypress.json file.
3. **Write a test**: Create a test file in the cypress/integration directory.
4. **Run tests**: Run next dev and then cypress open.

Example:

JavaScript

```
// cypress/integration/app.spec.js
describe('Navigation', () => {
  it('should navigate to the about page', () => {
    cy.visit('http://localhost:3000/');
    cy.get('a[href*="/about"]').click();
    cy.url().should('include', '/about');
    cy.get('h1').contains('About');
  });
});
```

88. How to debug Next.js apps in VS Code?

Debugging Next.js in VS Code is straightforward.

1. **Add a launch configuration**: Create a .vscode/launch.json file.
2. **Add the Node.js configuration**:
3. JSON

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Next.js: debug",
      "type": "node",
      "request": "launch",
      "runtimeExecutable": "npm",
      "runtimeArgs": ["run", "dev"],
      "skipFiles": ["<node_internals>/**"],
      "console": "integratedTerminal"
    }
```

```
  ]
}
```

4.
5.
6. **Start debugging**: Place breakpoints in your code, then press `F5` or click the "Run and Debug" button in VS Code.

89. How to analyze performance with `next build` and `next analyze`?

- **`next build`**: This command compiles your application for production. It generates a summary of your pages and their sizes, showing you which pages are statically rendered, server-rendered, etc. This is the first step to understand your build.
- **`@next/bundle-analyzer`**: This is a powerful tool to visualize the contents of your JavaScript bundles. It shows you which packages are contributing most to your bundle size, helping you identify opportunities for optimization.
    1. Install it: `npm install -D @next/bundle-analyzer`.
    2. Configure it in `next.config.js`.
    3. Run the build with a flag to analyze the bundles.

90. Best practices for testing Next.js apps.

1. **Start with integration tests**: They provide the most value for the least effort. Test user flows, not just individual functions.
2. **Test from a user's perspective**: Use React Testing Library's `screen.getByRole` to find elements as a user would.
3. **Mock dependencies**: Don't make real API calls in your tests. Mock them to make tests faster and more reliable.
4. **Cover critical flows with E2E tests**: Use Cypress or Playwright to ensure your most important user journeys (e.g., checkout, login) are working.
5. **Use a consistent testing strategy**: Decide on a testing framework and a set of conventions for your team.

---

10. Deployment & Real-World

91. How to deploy a Next.js app on Vercel?

Deploying to Vercel is the **easiest and most recommended way** to deploy a Next.js app, as Next.js and Vercel are both developed by the same team.

1. **Create a Vercel account** and connect it to your Git provider (GitHub, GitLab, etc.).
2. **Import your project**: Vercel will automatically detect that it's a Next.js app and configure the build settings.
3. Push to main branch: Every push to the main branch triggers a new production deployment. Every pull request gets a preview deployment.
   Vercel automatically handles SSR, SSG, and API routes as serverless functions.

92. How to deploy a Next.js app on Netlify?

Netlify also offers excellent support for Next.js.

1. **Install the Netlify CLI**: `npm install -g netlify-cli`.
2. **Install the Next.js build plugin**: `npm install @netlify/plugin-nextjs`.
3. **Link to your project**: `netlify init`.
4. Deploy: `netlify deploy --prod`.
   The build plugin handles the conversion of Next.js's serverless functions into a format compatible with Netlify's platform.

93. How to deploy a Next.js app on AWS/GCP/Azure?

You can deploy a Next.js app on any cloud provider, but it requires more manual setup.

- **SSR/API Routes**: You need to run a Node.js server.
  - **AWS**: Use a combination of **EC2** (for a traditional server), **ECS/EKS** (for containers), or **AWS Amplify** for a more managed solution.
  - **GCP**: Use **Google App Engine** or **Google Cloud Run** for a serverless container solution.
  - **Azure**: Use **Azure App Service** or **Azure Static Web Apps**.
- **SSG**: You can simply build the app and upload the `out/` directory to an S3 bucket or any static hosting service.

94. What is the difference between static export and server rendering for deployment?

- **Static Export**: You run `next build && next export` to produce a folder of static files. These files can be served from any static host or a CDN. This is for sites that **don't have SSR or API routes**. It's the simplest and most performant deployment method for static content.
- **Server Rendering**: This is the default. You run `next build && next start`. It requires a Node.js server to be running to handle incoming requests and render pages on the fly. This is necessary for applications that use `getServerSideProps` or API routes. The serverless function model on Vercel is a specific implementation of this.

95. What is Dockerizing a Next.js app?

Dockerizing a Next.js app means packaging it into a Docker container. This creates a self-contained, portable, and consistent environment that runs your application.

1. **Create a `Dockerfile`**: This file contains instructions for building the image (e.g., installing dependencies, copying files, running the build).
2. **Build the image**: `docker build -t my-next-app .`.
3. **Run the container**: `docker run -p 3000:3000 my-next-app`.

Why Dockerize? It ensures your app runs the same way in development, testing, and production, eliminating "it works on my machine" problems.

96. How to use PM2 with Next.js?

PM2 is a production process manager for Node.js applications. It helps you manage, run, and keep your Node.js app alive.

1. **Install PM2**: `npm install -g pm2`.
2. Start your app with PM2: pm2 start npm --name "my-next-app" -- run start.
   PM2 will automatically restart your app if it crashes and can manage multiple instances for load balancing. This is useful for self-hosted deployments.

97. What is CI/CD in Next.js? How to implement?

CI/CD (Continuous Integration/Continuous Deployment) is a set of practices that automates the build, test, and deployment process.

1. **CI (Continuous Integration)**: A build server (e.g., GitHub Actions, GitLab CI/CD) automatically runs tests every time a developer pushes code. If tests fail, it prevents the code from being merged.
2. **CD (Continuous Deployment)**: Once tests pass, the build server automatically deploys the code to a staging or production environment.

Implementation with GitHub Actions:

Create a .github/workflows/main.yml file that defines the steps: check out code, install dependencies, run tests (npm test), and deploy to a provider like Vercel or Netlify.

98. Difference between development and production builds in Next.js.

- **Development (`next dev`)**: The build is optimized for a fast developer experience.
  - **Hot Module Replacement (HMR)**: Changes are instantly reflected in the browser without a full reload.
  - **Warning/Error messages**: Verbose and helpful.
  - **Performance**: Not optimized; un-minified code.
- **Production (`next build && next start`)**: The build is optimized for performance and end-user experience.
  - **Minification**: Code is minified to reduce file size.
  - **Tree-shaking**: Unused code is removed.
  - **Caching**: Code splitting and prefetching are optimized for performance.
  - **Strict error handling**: Errors are minimized.

99. How to monitor performance of a deployed Next.js app?

- **Vercel Analytics**: If deployed on Vercel, you get built-in analytics for Core Web Vitals, page views, and more.
- **Web Vitals**: Use `next/web-vitals` to track key performance metrics like LCP, FID, and CLS. Send them to an analytics service.
- **Google Analytics/New Relic/Sentry**: Use these services to track user behavior, monitor errors, and gain insights into your application's performance.

100. Best practices for building scalable Next.js applications.

1. **Prioritize SSG/ISR**: Use static generation as the default for pages that don't need real-time data.
2. **Leverage the `app` directory**: Adopt the new features like React Server Components for improved performance and architecture.
3. **Use a CDN**: Vercel and Netlify use CDNs by default. For self-hosted, use a service like Cloudflare or Amazon CloudFront.
4. **Decouple API Routes**: For complex applications, consider using a separate, dedicated backend for heavy lifting to keep your Next.js app as a thin, performant layer.
5. **Modularize your code**: Break down your application into reusable components, services, and libraries.
6. **Use a managed database**: Use a scalable database solution like MongoDB Atlas or a managed PostgreSQL service instead of self-hosting.