React Questions - Week 6 (18Aug-25Aug)

Compiled by Ⓚ Kumar Nayan

❓ Q #1 (🟢 Basic)

Tags: 🧩 General Interview , 🧪 Testing

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is unit testing in the context of a React application, and what is its primary goal?

✅ **Answer:**

Unit testing is a method where the smallest isolatable parts of your code, known as "units," are tested in complete isolation from the rest of the application. In React, a unit is typically a single component, function, or hook.

The primary goal is to verify that each unit of the application works as expected on its own. These tests are fast to run and easy to debug because a failure points to a specific, isolated piece of code.

❓ Q #2 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Explain the difference between unit testing and integration testing. When would you choose one over the other?

✅ **Answer:**

Unit testing focuses on a single, small piece of code in isolation (e.g., a `Button` component), often using mocks to fake dependencies. It verifies that the individual part works correctly.

Integration testing checks how multiple units work together (e.g., a login form, an API service, and a user database). Its goal is to verify that different modules interact correctly and the system behaves as intended as a whole.

You would use unit tests for verifying the logic of individual components and functions quickly. You'd use integration tests to catch bugs that only appear when different parts of your system interact, which unit tests would miss.

❓ Q #3 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

You're converting a React testing suite from Jest to Vitest. What are the main code changes you'd expect to make, based on a simple component test?

✅ **Answer:**

For a basic component test, the transition from Jest to Vitest is minimal, as Vitest was designed with a Jest-compatible API. The main change is how you import testing functions and create mocks.

Instead of relying on Jest's globals, you explicitly import `describe`, `it`, `expect`, and `vi` from `'vitest'`. The mock function `jest.fn()` becomes `vi.fn()`. The rest of the code, including imports from React Testing Library like `render` and `fireEvent`, remains identical.

```jsx
// Jest code
const handleClick = jest.fn();

// Vitest code
import { vi } from "vitest";
const handleClick = vi.fn();
```

❓ Q #4 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🧪 Testing, 🏗️ Architecture

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Describe the "Testing Pyramid" concept using the test types mentioned in your notes. How would you allocate your testing budget across unit, integration, and E2E tests?

✅ **Answer:**

The Testing Pyramid is a strategy that guides test allocation. It suggests having a large base of fast, cheap **unit tests**, a smaller middle layer of **integration tests**, and a very small top layer of slow, expensive **end-to-end (E2E) tests**.

- **Unit Tests (Base):** Hundreds of tests for individual components and functions. They are fast, stable, and easy to maintain.
- **Integration Tests (Middle):** A moderate number of tests (e.g., 20) to check interactions between modules, like a component calling an API service.
- **E2E Tests (Peak):** Very few tests (e.g., 5) for critical user flows like login, checkout, or payment. They are slow, brittle, and hard to maintain but provide high confidence that the whole system works.

This structure ensures broad coverage at the unit level while verifying critical flows without the high cost and flakiness of excessive E2E tests.

❓ Q #5 (🟡 Intermediate)

Tags: 🧩 General Interview, 🧪 Testing

🧠 **Question:**

What is End-to-End (E2E) testing, and what critical business value does it provide that unit and integration tests cannot?

✅ **Answer:**

End-to-end (E2E) testing is a methodology used to validate the complete flow of an application from the user's perspective, ensuring that all integrated components work together correctly in a production-like environment. It simulates a real user journey, such as logging in, adding an item to a cart, and checking out.

The critical value it provides is **high confidence in user flows**. While unit and integration tests verify that parts of the system work correctly in isolation or in small groups, E2E tests confirm that the entire system works together to fulfill a user's goal. This helps ensure that new features or code changes do not break existing, business-critical pathways.

❓ Q #6 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🔁 Rendering

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Compare and contrast Client-Side Rendering (CSR), Server-Side Rendering (SSR), and Static Site Generation (SSG). For each, describe an ideal use case.

✅ **Answer:**

- **CSR (Client-Side Rendering):** The browser loads a minimal HTML shell and a JavaScript bundle. React then builds the UI in the browser. It offers a fast, app-like feel after the initial load but suffers from slow initial page load and poor SEO.
    - **Use Case:** A highly interactive, behind-a-login dashboard where SEO is not a concern (e.g., a project management tool).
- **SSR (Server-Side Rendering):** The server generates the full HTML for a page on each request and sends it to the browser. This provides excellent SEO and a fast First Contentful Paint (FCP) because the content is immediately visible.
    - **Use Case:** An e-commerce site or a news feed where content is dynamic and SEO is critical.

- **SSG (Static Site Generation):** The entire site's HTML is pre-built at build time. The server sends static files, making it incredibly fast and great for SEO. However, content can become stale and requires a rebuild to update.
    - **Use Case:** A documentation site, a portfolio, or a blog where content changes infrequently.

❓ Q #7 (🟣 Expert)

Tags: 🚀 MAANG-level , 🔁 Rendering , ⚙️ Performance

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What is Incremental Static Regeneration (ISR), and how does it solve the core limitation of traditional Static Site Generation (SSG)?

✅ **Answer:**

Incremental Static Regeneration (ISR) is a hybrid rendering strategy specific to Next.js that combines the benefits of SSG (speed, low server load) with the ability to update content dynamically.

It solves the core limitation of SSG—stale content—by allowing static pages to be regenerated *incrementally* in the background after a certain time interval (`revalidate`). When a user requests a page after its revalidation period has passed, they are served the stale (static) version, while Next.js triggers a regeneration in the background. The next user then receives the freshly generated page.

This provides the speed of static sites while ensuring content stays up-to-date without requiring a full site rebuild for every change.

```
jsxexport async function getStaticProps() {
 const res = await fetch('https://.../posts');
 const posts = await res.json();

 return {
   props: {
     posts,
   },
     // Next.js will attempt to re-generate the page:// - When a request comes in// - At most once every 60 seconds
   revalidate: 60,
 };
}
```

❓ Q #8 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🔁 `Rendering`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

In Next.js, explain the difference between `getStaticProps` and `getServerSideProps`. When would you absolutely need to use `getServerSideProps`?

✅ **Answer:**

- `getStaticProps` **(SSG):** This function runs at **build time**. It fetches data and pre-renders a page into a static HTML file. The same HTML is served to every user from a CDN, making it extremely fast. It's ideal for content that doesn't change often, like a blog post or documentation.
- `getServerSideProps` **(SSR):** This function runs on **every request** on the server. It fetches data and generates HTML dynamically for each incoming request. This ensures the content is always fresh.

You would absolutely need to use `getServerSideProps` when:

1. **Data must be real-time or changes constantly**, like a stock ticker or a live news feed.
2. **The page content is personalized per user** and must be rendered on the server for SEO or security reasons, such as a user's account dashboard.

❓ Q #9 (🟣 Expert)

Tags: 🚀 `MAANG-level` , 🔁 `Rendering`

🧠 **Question:**

Explain the roles of `getStaticPaths` and `getStaticProps` when creating dynamic routes in Next.js, like `/blog/[id]`. Walk through the process from build to runtime.

✅ **Answer:**

When using SSG for dynamic routes, `getStaticPaths` and `getStaticProps` work together during the build process.

1. `getStaticPaths` **(Runs first at build time):** Its job is to tell Next.js which specific paths (which `id`s) to pre-render. It fetches all possible items (e.g., all post IDs) and returns a `paths` array. For each object in this array, Next.js knows it needs to generate a page.
2. `getStaticProps` **(Runs for each path at build time):** After getting the list of paths, Next.js calls `getStaticProps` for each individual path. It receives the `params` (like `{ id: '1' }`) for that path, fetches the specific data for that single item, and passes it as props to the page component to generate the HTML.

At runtime: When a user visits `/blog/1`, the browser instantly receives the pre-generated HTML file created during the build, resulting in a very fast load time.

```
jsxexport async function getStaticPaths() {
    // Returns: { paths: [ { params: { id: '1' } }, { params: { id: '2' } } ], fallback: false }
}

export async function getStaticProps({ params }) {
    // Receives params.id, fetches data for that post// Returns: { props: { post } }
}
```

❓ Q #10 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧰 Debugging , 🔍 Real-World Scenario

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

Your QA team reports a bug that they cannot reproduce, but it's being reported by real users. How can a frontend monitoring tool like Sentry help you debug this "client-only" crash?

✅ **Answer:**

A monitoring tool like Sentry is essential for debugging client-only crashes. It acts as a "black box recorder" for your application in the user's browser.

When a crash occurs on a user's device, Sentry automatically captures and sends a detailed error report to your dashboard. This report includes:

1. **Stack Trace:** The exact line of code where the error occurred.
2. **Breadcrumbs:** A log of events that led up to the error, such as user clicks, route changes, API calls, and console logs. This provides context on how the

user triggered the bug.

3. **Environment Data:** Information about the user's browser, OS, and device, which can help identify environment-specific issues.

This allows the developer to see exactly what went wrong and how, without needing the QA team to manually reproduce the steps.

❓ Q #11 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🧰 Debugging

🧠 **Question:**

What are "Breadcrumbs" in the context of error tracking with Sentry, and how can you add custom breadcrumbs to enrich your error reports?

✅ **Answer:**

Breadcrumbs are a trail of events that occurred in your application leading up to an error. They provide a chronological log of user actions, application state changes, and other events, helping developers understand the context of a crash. Sentry automatically records many events like clicks, route changes, and API calls.

You can add **custom breadcrumbs** to log application-specific events using `Sentry.addBreadcrumb()`. This is useful for tracking important steps in a user flow, like adding an item to a cart or initiating a payment. This makes the error report much richer, as it shows the exact business logic steps the user took before the crash.

```
jsximport * as Sentry from "@sentry/react";

function checkout() {
  Sentry.addBreadcrumb({
    category: "cart",
    message: "User clicked checkout",
    level: "info",
  });

  try {
      // Some failing code
    throw new Error("Payment gateway timeout");
  } catch (err) {
    Sentry.captureException(err);
  }
}
```

❓ Q #12 (🟡 Intermediate)

Tags: 🧩 General Interview , 🏗️ Architecture

🧠 **Question:**

What is Docker, and what is the core problem it solves for development teams?

✅ **Answer:**

Docker is a tool that packages an application and all its dependencies—such as the code, runtime (e.g., Node.js), system tools, and libraries—into a single, isolated unit called a **container**.

The core problem it solves is the **"it works on my machine"** issue. By containerizing an application, Docker ensures that it runs exactly the same way on any machine, whether it's a developer's laptop, a testing server, or a production environment. This eliminates inconsistencies caused by differences in operating systems, software versions, or configurations.

❓ Q #13 (🟡 Intermediate)

Tags: 🧩 General Interview , 🏗️ Architecture , 💥 Pitfall

🧠 **Question:**

From a developer's perspective, what are the key benefits of using a Docker-based environment over a traditional local development setup?

✅ **Answer:**

A Docker-based environment offers several key advantages over a traditional local setup:

- **Consistency:** It eliminates environment drift. The app runs the same everywhere, preventing issues from different OS or library versions.
- **Easy Setup:** New developers can get started with a single command (`docker-compose up`) instead of manually installing and configuring databases, runtimes, and other tools.
- **Isolation:** Containers are isolated from the host machine, so dependencies for one project won't conflict with another.

- **Reproducibility:** The environment is defined in code (`Dockerfile`, `docker-compose.yml`), making it easy to tear down and perfectly recreate at any time.

❓ Q #14 (🔴 Advanced)

Tags: 🚀 `MAANG-level`, 🏗️ `Architecture`

🧠 **Question:**

Describe the purpose of a multi-stage `Dockerfile` in the context of building a production-ready React application.

✅ **Answer:**

A multi-stage `Dockerfile` is a best practice for creating optimized, lightweight production images. For a React app, it separates the **build environment** from the **production environment**.

- **Stage 1 (Build Stage):** This stage uses a heavy base image with all the build tools needed, like a full Node.js image. It copies the source code, installs all dependencies (including `devDependencies`), and runs the production build command (e.g., `npm run build`). The output is a folder of static assets (e.g., `/build`).
- **Stage 2 (Production Stage):** This stage starts with a very lightweight, secure base image, like Nginx or Alpine. It then copies **only the static assets** from the build stage. It doesn't contain the source code, `node_modules`, or any build tools.

This results in a final Docker image that is significantly smaller and has a reduced attack surface, making it more secure and faster to deploy.

```
jsx# Stage 1: Build React app
FROM node:18 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Stage 2: Serve with Nginx
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

❓ Q #15 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture , 🔍 Real-World Scenario

🧠 **Question:**

How would you use `docker-compose` to orchestrate a full-stack application consisting of a React frontend, a Node.js/Express backend, and a MongoDB database?

✅ **Answer:**

`docker-compose` is used to define and run multi-container Docker applications. For a full-stack app, you would create a `docker-compose.yml` file to define three services: `frontend`, `backend`, and `db`.

1. `frontend` **service:** This would build the React app's `Dockerfile` (likely using a multi-stage build to serve static files with Nginx) and map a host port (e.g., 3000) to the container's port (e.g., 80).
2. `backend` **service:** This would build the Node.js/Express app's `Dockerfile`, expose its port (e.g., 5000), and use environment variables to pass the database connection string.
3. `db` **service:** This would use a pre-built public image, like `mongo:6`, and use volumes to persist database data across container restarts.

The services would be linked using `depends_on` to control startup order, ensuring the database is ready before the backend starts, and the backend is ready before the frontend. A single `docker-compose up` command would then build and run the entire stack.

❓ Q #16 (🟡 Intermediate)

Tags: 🧩 General Interview , 🏗️ Architecture

🧠 **Question:**

What is CI/CD, and why is it important in a modern development workflow?

✅ **Answer:**

CI/CD stands for Continuous Integration and Continuous Deployment/Delivery.

- **Continuous Integration (CI):** This is the practice of developers merging their code changes into a central repository frequently. Each merge triggers an automated build and test sequence. The goal is to catch bugs early and prevent integration problems.
- **Continuous Deployment (CD):** This is the practice of automatically deploying all code changes that pass the CI stage to a testing or production environment. This makes the release process faster and more reliable.

CI/CD is important because it automates the build, test, and deployment process, leading to faster iteration, higher code quality, and more consistent and reliable releases.

❓ Q #17 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🏗️ `Architecture`

🧠 **Question:**

Explain how you would set up a GitHub Actions workflow to run linting, formatting checks, and tests on every pull request to the `main` branch.

✅ **Answer:**

To set up this workflow, you would create a YAML file (e.g., `ci.yml`) inside the `.github/workflows` directory of your repository.

The workflow would be configured as follows:

- `on: pull_request:` This trigger would make the workflow run on every pull request targeting specified branches like `main`.
- `jobs:` A `build` job would be defined to run on a specified runner, like `ubuntu-latest`.
- `steps:`
    1. `actions/checkout@v4`: Checks out the repository code.
    2. `actions/setup-node@v4`: Sets up the specified Node.js version and configures dependency caching.
    3. `npm ci`: Installs dependencies cleanly from the `package-lock.json` file.
    4. `npm run lint`: Runs the linter (e.g., ESLint).

5. `npm run format:check`: Runs a format checker (e.g., Prettier) to ensure code style is consistent.
6. `npm test`: Runs the test suite (e.g., Vitest or Jest).

If any of these steps fail, the workflow fails, and the pull request is marked with a red check, preventing a merge of broken or improperly formatted code.

❓ Q #18 (🟡 Intermediate)

Tags: 🧩 General Interview , 🏗️ Architecture

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What are the main advantages of using a deployment platform like Vercel or Netlify over deploying a React app manually?

✅ **Answer:**

Using a deployment platform like Vercel offers significant advantages over manual deployment:

- **Automation:** They integrate directly with your Git repository (e.g., GitHub) and automatically build and deploy your application whenever you push changes to a specific branch.
- **CI/CD Built-in:** The entire process of pulling code, installing dependencies, building, and deploying is handled automatically, ensuring consistency.
- **Preview Deployments:** A unique, live preview URL is automatically generated for every pull request, allowing for easy review of changes before merging.
- **No Server Management:** You don't have to manage servers, SSL certificates, or DNS configurations. The platform handles all of this.
- **Instant Rollbacks:** You can quickly and easily revert to a previous deployment with a single click if a bug is discovered.

❓ Q #19 (🟢 Basic)

Tags: 🧩 General Interview , 🏗️ Architecture

🧠 **Question:**

How does a platform like Vercel handle deployments for a Next.js or React application? Describe the typical workflow.

✅ **Answer:**

Vercel provides a seamless, Git-based workflow for deploying frontend applications.

1. **Connect Repo:** You connect your GitHub, GitLab, or Bitbucket repository to a Vercel project.
2. **Deploy:** Vercel automatically detects the framework (like Next.js), builds the project with the correct settings, and deploys it.
3. **Get Live URL:** After the first deployment, you get a live URL (e.g., `my-app.vercel.app`).
4. **Auto Redeploy:** Whenever you push new code to your production branch (e.g., `main`), Vercel automatically triggers a new build and deployment, updating your live site with the changes.

❓ Q #20 (🟢 Basic)

Tags: 🧩 `General Interview` , 🧪 `Testing`

🧠 **Question:**

In a Vitest or Jest test file, what is the purpose of the `describe` and `it` blocks?

✅ **Answer:**

- `describe`**:** This function is used to create a **test suite**, which is a way to group related tests together. It helps organize your test file into logical sections, making the test output more readable. For example, you could have a `describe('Button Component', ...)` block to contain all tests for the Button component.
- `it`**:** This function (or its alias, `test`) defines an **individual test case**. It describes a specific behavior or scenario that you want to verify. Each `it` block should ideally test one single thing and contain the "Arrange-Act-Assert" steps for that scenario.

```
jsxdescribe('Button Component', () => {
  it('should render with default text', () => {
    // Test logic here
  });

  it('should be disabled when the disabled prop is true', () => {
    // Test logic here
  });
});
```

❓ Q #21 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

In React Testing Library, what is the role of `render`, `screen`, and `fireEvent`?

✅ **Answer:**

- `render`: This function renders a React component into a virtual DOM that you can test. It's the first step in any component test.
- `screen`: This is an object that holds various queries to find elements on the "screen" (the virtual DOM). It encourages user-centric queries, like `screen.getByText('Click Me')` or `screen.getByRole('button')`, which tests the component the way a user would interact with it.
- `fireEvent`: This function is used to simulate user events, such as clicks, input changes, or form submissions. For example, `fireEvent.click(buttonElement)` simulates a user clicking a button, allowing you to test the component's interactive behavior.

❓ Q #22 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🧪 Testing , 💥 Pitfall

🧠 **Question:**

E2E tests are often described as "flaky." What does this mean, and what are some common causes of flakiness in a Cypress test?

✅ **Answer:**

A "flaky" test is one that passes sometimes and fails at other times without any changes to the code. This makes them unreliable and frustrating to deal with.

Common causes of flakiness in E2E tests include:

- **Timing Issues:** The test might try to interact with an element before it has fully loaded, appeared, or become interactive due to an animation or an API call.
- **Network Dependency:** The test might fail if a backend API is slow to respond or temporarily unavailable.
- **Unpredictable UI State:** The application state might not be reset properly between tests, causing one test to affect the outcome of another.
- **Asynchronous Operations:** The test script doesn't properly wait for asynchronous operations (like data fetching) to complete before making assertions.

Cypress has built-in waits and retries to help mitigate this, but robust tests often require explicit waits or assertions to ensure the application is in the correct state before proceeding.

❓ Q #23 (🟢 Basic)

Tags: 🧩 General Interview , 📦 State Management

🧠 **Question:**

What is rendering in React, and why is it a fundamental concept to understand?

✅ **Answer:**

Rendering is the process of React turning your declarative component code (JSX) into actual UI that can be displayed in a browser. It involves creating DOM nodes and updating them in response to changes in state or props.

It's a fundamental concept because understanding *when* and *why* your components render is crucial for managing application state, optimizing performance, and debugging unexpected behavior. Unnecessary re-renders are a common source of performance bottlenecks in React applications.

❓ Q #24 (🟢 Basic)

Tags: 🧩 General Interview , 🧰 Debugging

🧠 **Question:**

What are two popular frontend monitoring tools, and what is a key difference between them?

✅ **Answer:**

Two popular frontend monitoring tools are **Sentry** and **LogRocket**.

- **Sentry** excels at **error tracking and performance monitoring**. It captures detailed stack traces, breadcrumbs, and performance metrics to help developers quickly diagnose and fix issues. It's like a "black box recorder" for your app's errors.
- **LogRocket** focuses on **session replay**. It records videos of user sessions, allowing you to watch a pixel-perfect replay of what a user did leading up to a bug.

The key difference is that Sentry provides deep, technical error reports, while LogRocket provides a visual, user-centric context by replaying their exact interactions.

❓ Q #25 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture , 🔍 Real-World Scenario

🧠 **Question:**

You are designing a system with multiple React frontend applications that all need to communicate with the same backend services. How would you architect the deployment and traffic management for this system?

✅ **Answer:**

A scalable architecture for this scenario would involve decoupling the frontends from the backend.

1. **Frontend Deployment:** Each React application would be deployed independently on a platform optimized for frontends, like Vercel or Netlify. This allows each team to iterate and deploy at its own pace.

2. **Backend Deployment:** The backend services (e.g., Node.js APIs) and the database would be deployed on a platform suited for backends, like Railway or Fly.io.
3. **Centralized API Gateway/Load Balancer:** All React apps would send requests to a single, central API endpoint (e.g., `api.myapp.com`). A load balancer at this endpoint would distribute the incoming traffic across multiple instances of the stateless backend servers.
4. **Stateless Backend:** The backend servers should be stateless, relying on a shared database or cache for storing state. This allows the system to be scaled horizontally by simply adding more backend instances as traffic grows.

This architecture ensures that the frontend and backend can be scaled and deployed independently, improving maintainability and resilience.

❓ Q #26 (🟣 Expert)

Tags: 🚀 `MAANG-level`, 🔁 `Rendering`

🧠 **Question:**

In Next.js, what is the difference between `fallback: 'blocking'` and `fallback: true` within `getStaticPaths`?

✅ **Answer:**

Both options are used for pages that were not pre-generated at build time via Static Site Generation (SSG).

- `fallback: true`: When a user requests a non-pre-rendered page, Next.js immediately serves a "fallback" version of the page (e.g., a loading skeleton). In the background, it generates the full page and caches it. The page will then re-render with the full data. This is a non-blocking approach.
- `fallback: 'blocking'`: When a user requests a non-pre-rendered page, the user's browser "waits" while the server generates the HTML on the fly. The user is served the fully generated page on the first visit, and it is then cached for subsequent visitors. There is no intermediate loading state visible to the user, but the initial load time is longer.

❓ Q #27 (🟡 Intermediate)

Tags: 🧩 General Interview , 🏗️ Architecture

🧠 **Question:**

In a CI/CD pipeline defined in GitHub Actions, why is `npm ci` recommended over `npm install` for installing dependencies?

✅ **Answer:**

`npm ci` is recommended for CI/CD environments for two main reasons:

1. **Reproducibility:** `npm ci` performs a "clean install" by deleting the existing `node_modules` folder and installing dependencies strictly based on the versions specified in the `package-lock.json` file. This ensures the build is always identical and reproducible. `npm install` can update the lock file, leading to potential inconsistencies.
2. **Speed:** Because `npm ci` doesn't need to resolve dependency versions and follows the lock file exactly, it is often significantly faster than `npm install`, which is a key advantage in automated pipelines.

❓ Q #28 (🟢 Basic)

Tags: 🧩 General Interview , 🏗️ Architecture

🧠 **Question:**

What is the fundamental difference between a Docker Image and a Docker Container?

✅ **Answer:**

The relationship between an image and a container can be compared to a recipe and a cooked meal.

- An **Image** is a read-only, inert template or blueprint that packages up the application code, a runtime, dependencies, and configurations.
- A **Container** is a live, running instance of an image. You can start, stop, and interact with a container. It's the actual "sealed box" where the application runs.

You build an image once, and you can run many containers from that single image.

❓ Q #29 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧰 Debugging

🧠 **Question:**

How does Sentry's Error Boundary component help you catch and report UI rendering errors in a React application?

✅ **Answer:**

A `Sentry.ErrorBoundary` is a React component that you wrap around parts of your application, typically the root `<App />` component. It acts as a safety net. If any component within the boundary throws a JavaScript error during rendering, the boundary will "catch" it instead of letting the entire application crash.

When it catches an error, it does two things:

1. **Reports the Error:** It automatically captures the error and sends a detailed report to your Sentry dashboard.
2. **Renders a Fallback UI:** It displays a user-friendly fallback component, so the user doesn't see a broken white screen.

```
jsximport * as Sentry from "@sentry/react";

<Sentry.ErrorBoundary fallback={<p>Something went wrong</p>}>
  <App />
</Sentry.ErrorBoundary>
```

❓ Q #30 (🟢 Basic)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

What is the recommended naming convention for Cypress test files, and what is the reason for this recommendation?

✅ **Answer:**

Cypress recommends naming test files with a `.cy.js` or `.cy.ts` extension (e.g., `login.cy.js`).

The reason for this convention is to **clearly distinguish Cypress end-to-end tests** from other types of tests in your project, such as unit tests which commonly use a `.test.js` or `.spec.js` suffix (e.g., `App.test.js`). This helps in organizing the test suite and configuring different test runners to pick up the correct files.

❓ Q #31 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🔁 `Rendering` , 🔍 `Real-World Scenario`

🧠 **Question:**

Why is it necessary to use a production-like environment or a platform like Vercel to properly test Incremental Static Regeneration (ISR)?

✅ **Answer:**

ISR functionality, specifically the `revalidate` property, does not work in a standard development environment (`next dev`). This is because ISR relies on a production server and a caching layer (like a CDN) to operate correctly.

To test it, you need an environment that simulates production behavior:

- **Locally:** You must run `next build && next start`.
- **On a Platform:** Vercel is the ideal place to test it because its infrastructure is built to support Next.js features out-of-the-box. Vercel's CDN and serverless functions automatically handle the stale-while-revalidate logic, caching, and background regeneration that make ISR work.

❓ Q #32 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🧪 `Testing` , 🏗️ `Architecture`

🧠 **Question:**

What is the role of the `cypress.config` file in a Cypress project, and where should it be located?

✅ **Answer:**

The `cypress.config.js` (or `.ts`) file is the central configuration file for Cypress. Its primary role is to define global settings and behaviors for your entire test suite. You can configure things like the base URL, default viewport dimensions, environment variables, and timeouts.

It should be located in the **root directory** of your project, at the same level as `package.json`.

❓ Q #33 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔁 Rendering , ⚙️ Performance

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

What are the primary pros and cons of Server-Side Rendering (SSR)?

✅ **Answer:**

Pros:

- **Better SEO:** Because the server sends fully rendered HTML, search engine crawlers can easily index the content, which is crucial for public-facing sites.
- **Faster First Contentful Paint (FCP):** Users see content immediately because the browser receives ready-to-display HTML, improving perceived performance.

Cons:

- **Higher Server Load:** The server has to generate a page on every single request, which is computationally more expensive than just serving static files.
- **Slower Time to First Byte (TTFB):** Because the server must fetch data and render the page before sending a response, it can be slower to receive the very first byte of data compared to SSG.

❓ Q #34 (🟢 Basic)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

In a Vitest or Jest test, what is the purpose of a mock function created with `vi.fn()`?

✅ **Answer:**

A mock function created with `vi.fn()` (or `jest.fn()`) is a "spy" or a fake function. Its purpose is to replace a real function (like a prop passed to a component) during a test.

This allows you to test a component's behavior in isolation. You can check things like:

- Whether the function was called.
- How many times it was called.
- What arguments it was called with.

This is done without actually executing the original function's logic, which might have side effects like making an API call.

❓ Q #35 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🏗️ `Architecture`

🧠 **Question:**

From an architectural standpoint, why is it beneficial to deploy the frontend and backend of a full-stack application on separate, specialized platforms?

✅ **Answer:**

Deploying the frontend and backend separately allows each part to be optimized for its specific needs, leading to better performance, scalability, and maintainability.

- **Frontend Needs:** Frontends (like React apps) are composed of static files (HTML, CSS, JS). They benefit most from being served from a global **Content Delivery Network (CDN)**, which platforms like Vercel and Netlify provide. This ensures fast load times for users anywhere in the world.
- **Backend Needs:** Backends need to run continuously, handle secure business logic, and connect to a database. Platforms like Railway or Fly.io are designed

for this, offering persistent server environments, easy database provisioning, and the ability to scale compute resources independently from the frontend.

This separation of concerns allows frontend and backend teams to deploy independently and use the best tool for their respective jobs.

❓ Q #36 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

What value does the `@testing-library/jest-dom` package add to your tests?

✅ **Answer:**

The `@testing-library/jest-dom` package extends the built-in test matchers with a set of custom, DOM-specific matchers. This makes assertions much more declarative, readable, and focused on the state of the UI.

Instead of writing complex assertions to check for an element's existence or attributes, you can use simple matchers like:

- `.toBeInTheDocument()`
- `.toBeVisible()`
- `.toBeDisabled()`
- `.toHaveTextContent('some text')`

This improves the clarity and maintainability of your tests.

❓ Q #37 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture

🧠 **Question:**

In a `docker-compose.yml` file, what is the purpose of the `volumes` directive, and why is it critical for a database service?

✅ **Answer:**

The `volumes` directive in Docker Compose is used to persist data generated by and used by Docker containers.

It is critical for a database service because containers are ephemeral by default. If a database container is stopped or restarted, any data written to its filesystem is lost. By mapping a volume from the host machine to the directory inside the container where the database stores its files (e.g., `/data/db` for MongoDB), the data is saved on the host. This ensures that the database's data persists even if the container is removed and recreated.

```jsx
// docker-compose.yml
services:
  db:
    image: mongo:6
    volumes:
      - mongo_data:/data/db // Persists data in the 'mongo_data' volume

volumes:
  mongo_data:
```

❓ Q #38 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

What does it mean to run Cypress tests in "headless mode," and what is the primary use case for it?

✅ **Answer:**

Running Cypress in "headless mode" means executing the tests in a browser that runs in the background, without a visible UI. The browser window is not rendered on the screen.

The primary use case for headless mode is for **automated test execution in CI/CD pipelines**. Running `npx cypress run` triggers headless mode by default. It is faster and consumes fewer resources than running with a full graphical interface, making it ideal for servers and automated environments.

❓ Q #39 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🧰 Debugging , ⚙️ Performance

🧠 **Question:**

Explain the `tracesSampleRate` configuration in Sentry. How would you set it for a development environment versus a high-traffic production environment?

✅ **Answer:**

The `tracesSampleRate` configuration in Sentry controls the percentage of transactions that are captured for performance monitoring.

- **In a development environment:** You would typically set it to `1.0`, which captures 100% of transactions. This ensures you get performance data for every action you take while debugging.
- **In a high-traffic production environment:** Capturing 100% of transactions would be excessively noisy and costly. You should set it to a much lower value, such as `0.2` (for 20%) or less. This provides a statistically significant sample of performance data to identify trends and bottlenecks without overwhelming your system or your Sentry quota.

❓ Q #40 (🟡 Intermediate)

Tags: 🧩 `General Interview`, 🏗️ `Architecture`

🧠 **Question:**

In a `docker-compose.yml` file, what is the purpose of the `depends_on` key?

✅ **Answer:**

The `depends_on` key is used to control the startup and shutdown order of services defined in the `docker-compose.yml` file. If a service `frontend` depends on `backend`, Docker Compose will ensure that the `backend` service is started *before* the `frontend` service starts.

This is crucial for preventing race conditions, such as a backend application trying to connect to a database that hasn't started yet.

❓ Q #41 (🟡 Intermediate)

Tags: 🧩 `General Interview`, 🧪 `Testing`, 💥 `Pitfall`

🔥 **EXTREMELY IMPORTANT**

🧠 **Question:**

React Testing Library promotes using "user-centric" queries. What does this mean, and why is it considered a best practice for writing resilient tests?

✅ **Answer:**

"User-centric" queries find elements on the page in the same way a user would. This means querying by visible text (`getByText`), accessibility role (`getByRole`), or label text (`getByLabelText`).

This is a best practice because it decouples tests from implementation details. If you query by a `data-testid` or a CSS class, your test will break if a developer refactors the code and changes that attribute, even if the user experience remains identical. By testing from the user's perspective, your tests are more resilient to refactoring and are more likely to fail only when the user-facing behavior actually breaks.

❓ Q #42 (🟢 Basic)

Tags: 🧩 General Interview , 🏗️ Architecture , 💥 Pitfall

🧠 **Question:**

What are some of the main drawbacks of deploying a web application manually?

✅ **Answer:**

Manually deploying a web application is prone to human error and inconsistency. The main drawbacks are:

- **Errors and Inconsistency:** The process can fail due to environment differences (the "it works on my machine" problem).
- **Painful Rollbacks:** Reverting to a previous version is a difficult and slow manual process.
- **No Automated Checks:** There is no automated testing or quality gate before the code goes live.
- **Lack of Collaboration:** Different team members might deploy in slightly different ways, leading to unpredictable outcomes.

❓ Q #43 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔁 Rendering

🧠 **Question:**

In the context of Server-Side Rendering (SSR) or Static Site Generation (SSG), what is "hydration"?

✅ **Answer:**

Hydration is the client-side process where React "attaches" itself to the static HTML that was rendered by the server. When the browser first receives the HTML from the server, the page is visible but not interactive.

The JavaScript bundle then runs, and React walks the existing HTML tree, attaches the necessary event listeners (like `onClick`), and initializes the application state. This process makes the static page a fully interactive single-page application.

❓ Q #44 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture

🧠 **Question:**

Why would you choose a platform like Railway or Fly.io for a backend service instead of a platform like Vercel?

✅ **Answer:**

While Vercel is excellent for frontends and can run serverless functions, platforms like Railway and Fly.io are purpose-built for hosting backends and databases. The key reasons to choose them for a backend are:

- **Persistent Services:** They are designed to run long-running server processes, which is necessary for most backend APIs.
- **Database Hosting:** They offer easy, integrated database provisioning and management.
- **Greater Control:** They provide more control over the runtime environment, scaling, and networking, which is often required for complex backend applications.

Vercel is optimized for serving static assets and running short-lived serverless functions, making it less suitable for a persistent backend server.

❓ Q #45 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

What is the purpose of the `support` folder in a standard Cypress project structure?

✅ **Answer:**

The `support` folder in a Cypress project is for housing reusable behaviors and global configurations that you want to apply to all of your tests. It typically contains two key files:

- **`e2e.js`:** This file is executed before every single test file. It's the ideal place to import custom commands or add global `beforeEach` hooks.
- **`commands.js`:** This is where you define custom Cypress commands (e.g., `Cypress.Commands.add('login', ...)`) to abstract away repetitive sequences of actions.

❓ Q #46 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture

🧠 **Question:**

What does the `EXPOSE` instruction in a Dockerfile do, and does it publish the container's port to the host?

✅ **Answer:**

The `EXPOSE` instruction is a form of documentation that signals which port(s) a container listens on at runtime. It serves as metadata for developers and tools to understand the container's networking.

However, `EXPOSE` **does not** publish the port. It does not make the port accessible from the host machine. To actually map a container's port to a host port, you must use the `-p` flag with `docker run` or the `ports` mapping in a `docker-compose.yml` file.

❓ Q #47 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing , 💥 Pitfall

🧠 **Question:**

Why is debugging a failing end-to-end test generally more difficult than debugging a failing unit test?

✅ **Answer:**

Debugging an E2E test is harder because its scope is much larger. A failing E2E test involves **many moving parts**: the frontend code, the browser environment, network requests, backend APIs, and the database. The failure could be in any one of these components or in the interaction between them.

In contrast, a unit test is completely isolated. Its failure points directly to a specific, small piece of code (a single function or component), making the root cause much easier and faster to identify.

❓ Q #48 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧪 Testing

🧠 **Question:**

What is the recommended best practice for making the matchers from `@testing-library/jest-dom` available across all test files in a project?

✅ **Answer:**

The best practice is to import `@testing-library/jest-dom` **once** in a global test setup file. Most testing frameworks, like Jest and Vitest, have a configuration option (e.g., `setupFilesAfterEnv`) where you can specify a setup file that runs before all tests.

By placing `import '@testing-library/jest-dom';` in this central file, the custom matchers become available globally in every test file without needing to be imported repeatedly.

❓ Q #49 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🏗️ Architecture

🧠 **Question:**

In a CI workflow, what is the purpose of a command like `prettier --check .`, and how does it help maintain code quality?

✅ **Answer:**

The command `prettier --check .` scans the project files and checks if they are formatted according to the defined Prettier rules, but it **does not** modify them.

Its purpose in a CI workflow is to act as a **code style gatekeeper**. If any developer commits code that is not correctly formatted, this check will fail, causing the CI pipeline to fail. This prevents the pull request from being merged. It enforces a consistent code style across the entire codebase automatically, ensuring readability and maintainability without manual intervention.

❓ Q #50 (🟢 Basic)

Tags: 🧩 General Interview, 🏗️ Architecture

🧠 **Question:**

Briefly define the three core Docker concepts: Dockerfile, Image, and Container.

✅ **Answer:**

- **Dockerfile:** A text file with step-by-step instructions on how to build a Docker image. It's the recipe.
- **Image:** A read-only, lightweight, and standalone template that contains the application code, runtime, and dependencies. It's the packaged-up, ready-to-ship meal.
- **Container:** A running instance of an image. It's the actual execution of the recipe—the running application.

❓ Q #51 (🟡 Intermediate)

Tags: 🧩 General Interview, 💥 Pitfall, 🧰 Debugging

🧠 **Question:**

What fundamental problem in React do Error Boundaries solve?

✅ **Answer:**

By default, a JavaScript error in any part of the UI component tree will cause the entire React application to unmount, resulting in a blank white screen for the user.

Error Boundaries solve this problem. They are React components that **catch JavaScript errors** in their child component tree, **log those errors**, and **display a fallback UI** instead of letting the entire application crash. This provides a more graceful user experience and ensures that an error in one part of the UI doesn't take down the whole application.

❓ Q #52 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔁 Rendering , ⚙️ Performance , 💥 Pitfall

🧠 **Question:**

From a user experience perspective, what is the single biggest drawback of pure Client-Side Rendering (CSR)?

✅ **Answer:**

The single biggest drawback of CSR is the **slow initial load time**, which often manifests as a "blank screen" or a loading spinner. The browser has to download, parse, and execute a potentially large JavaScript bundle before any meaningful content can be rendered on the screen. This can lead to a poor user experience, especially on slower network connections or less powerful devices.