Rapi Apps
www.tryrapi,com
admin@tryrapi.com
5203 Juan Tabo Boulevard Northeast, Albuquerque, USA, 87111

Remix Resilience Developer Assessment

Role: Mid to Senior Remix Developer **Estimated Time:** 8 Hours **Submission Format:** A link to a public GitHub repository.

1. Project Scenario: The Unreliable Inventory Dashboard

You are building an internal Inventory Dashboard for a warehouse manager. The core challenge is that the backend API you must consume is **legacy, slow, and highly unreliable**.

The API exhibits the following critical faults:

1. **High Latency:** Fetching the main inventory list takes 3 seconds.
2. **Unreliability:** It throws random 500 errors 20% of the time.
3. **Slow Mutations:** Updating stock takes a noticeable amount of time.

Your mission is to demonstrate mastery of **Remix-specific architectural patterns** (Streaming, Optimistic UI, and Error Boundaries) to make the application feel **instant** and **unbreakable**, all while using the **Shopify Polaris component library** for the user interface.

2. Setup Instructions

1. **Start a Project:** Initialize a fresh Remix project using the stable template:

   ```
   npx create-react-router@latest inventory-test
   ```

2. **Install Polaris:** Install and configure the necessary Shopify Polaris packages (e.g., @shopify/polaris) and ensure the styles are correctly imported and rendered across your application.

3. **Implement the Backend Mock:** **You are forbidden from connecting a real database or modifying this mock file's logic.** Create a file at

app/models/inventory.server.ts and paste the following "Chaos Backend" code exactly.

Chaos Backend: **app/models/inventory.server.ts**

```
type Item = { id: string; name: string; stock: number };

// Mock Database (Simulates volatile data)
let MOCK_DB: Item[] = [
  { id: "1", name: "Super Widget A", stock: 10 },
  { id: "2", name: "Mega Widget B", stock: 0 },
  { id: "3", name: "Wonder Widget C", stock: 5 },
  { id: "4", name: "Hyper Widget D", stock: 2 },
];

/**
 * FETCH: Simulates a slow network request with a 20% chance of failure
 */
export async function getInventory(): Promise<Item[]> {
  // 1. Artifical delay (3 seconds)
  await new Promise((resolve) => setTimeout(resolve, 3000));

  // 2. Artificial Random Failure (20% chance)
  if (Math.random() < 0.2) {
    throw new Error("500: Random Legacy API Failure");
  }

  return MOCK_DB;
}

/**
 * MUTATION: Simulates a risky mutation (stock deduction)
 */
export async function claimStock(id: string) {
  // 1. Artificial delay (1 second)
  await new Promise((resolve) => setTimeout(resolve, 1000));

  const item = MOCK_DB.find((i) => i.id === id);

  if (!item) throw new Error("Item not found");
  if (item.stock <= 0) throw new Error("Out of stock");

  item.stock -= 1;
  return item;
}
```

3. The Required Tasks

Create a single route at /dashboard that fulfills the following three requirements:

Task 1: Eliminate the "White Screen" (Streaming & Performance)

If you simply await getInventory() in your Remix loader, the user stares at a blank screen for 3 seconds.

- **Requirement 1 (Immediate Render):** The page shell must render **immediately** (0ms delay).
- **Requirement 2 (Streaming):** The inventory list content must show a skeleton, or spinner while the 3-second getInventory data fetches in the background.

Task 2: Achieve Instant Feedback (Optimistic UI & Race Conditions)

Add a "Claim One" button next to each inventory item. Since the claimStock action takes 1 second, this creates a poor user experience.

- **Requirement 1 (Optimistic Update):** When the user clicks "Claim One", the stock number must decrease **instantly** (0ms delay) in the UI, before the server responds.
- **Requirement 2 (Rollback):** If the server eventually returns an error (e.g., *"Out of Stock"*), the UI must automatically **rollback** the stock count to the previous number.
- **Requirement 3 (Protection):** Use the appropriate Remix hook (useFetcher or useNavigation) to prevent users from accidentally double-submitting the form while the network request is pending.

Task 3: Contain the Blast (Route-Level Error Boundaries)

Since **getInventory** fails randomly 20% of the time, the app currently crashes the whole page.

- **Requirement 1 (Containment):** Implement a **Route-Level ErrorBoundary** so that if the inventory data fails to load, the main page structure must remain visible.

- **Requirement 2 (User Feedback):** Only the list area should show an error message. Use a Polaris **Banner** component to display the error and a **Button** to retry.
- **Requirement 3 (Resilience):** The "Retry Button" must re-run the **loader** (attempting the fetch again) without forcing a full browser page refresh.

4. Constraints & Evaluation Rubric

Your solution will be assessed based on the following criteria, with the focus being on **Remix fundamentals**, **problem-solving**, and adherence to the Shopify design system.

Area	**What We Are Looking For**	**Anti-Patterns (Will be marked down)**
Design System	**All UI elements** (tables, buttons, loading states, error messages) must use **Shopify Polaris components.**	Using standard HTML buttons, divs, or any other third-party styling library.
Performance	Correct use of defer() and <Await> to stream data.	Using useEffect() to fetch data client-side.
UX/State	Implementation of **Optimistic UI** using useFetcher state.	Using React's useState() to manually manage inventory count on the client.
Mutations	Proper use of HTML <Form> or fetcher.Form (Progressive Enhancement).	Using event.preventDefault() and manual fetch calls in an onClick.
Error Handling	Route-level ErrorBoundary exported from the route file.	Wrapping code in component-level try/catch blocks.

Submission

1. Commit all changes to your Git repository.
2. Push your code to a public repository (GitHub, GitLab, etc.).
3. Include a brief README.md that explains the key implementation choices for **Task 2 (Optimistic UI)** and **Task 3 (Retry Logic)**.
4. Send the repository link to the hiring team before the deadline.

Good luck!