Below is a **4-part test** designed to be completed in **4–5 hours** by a senior MERN-stack engineer. Each part exercises a specific skill area (MongoDB, Node.js, React, debugging & testing) and includes at least one twist or “mini-design” challenge that forces real thinking beyond AI boilerplate. Feel free to use AI assistance and tools but pay attention to the instructions and the context.

---

## □ Part 1: MongoDB Aggregation & Data Modeling (≈ 45 minutes)

> **Scenario:** You’re building analytics for a simple social-network prototype. Users can follow each other and make “posts.”
>
> **Dataset sample:**
>
> ```
>  // users
> { _id: ObjectId("u1"), name: "Alice", joined: ISODate("2024-01-15T09:00Z") }
> { _id: ObjectId("u2"), name: "Bob",   joined: ISODate("2024-02-02T12:30Z") }
> // follows
> { follower: "u1", following: "u2" }
> // posts
> { _id: ObjectId("p1"), author: "u2", content: "Hello!", created: ISODate("2024-03-10T18:00Z") }
> { _id: ObjectId("p2"), author: "u1", content: "Hey Bob", created: ISODate("2024-03-11T09:15Z") }
> ```

1. **Design your collection schemas** (briefly, in 5–7 lines) so that:
   - You can list each user’s followers (and who they follow) efficiently.
   - You can page through posts in reverse-chronological order.
2. **Write a single aggregation pipeline** that, for a given user ID, returns:
   - The **10 most recent posts** from that user’s followings, sorted newest → oldest.
   - Each with the post’s `content`, `created`, and the author’s `name`.
3. **Explain in 2–3 sentences** how you’d index these collections for maximum read performance.

---

## □ Part 2: Node.js API & Middleware (≈ 1 hour)

> **Scenario:** Extend the back-end for the social-network above. We need role-based access control (RBAC) for posts.

1. **Create an Express app** with:
   - `POST /login` that returns a signed JWT (no DB required—hard-code two users: `{id: "u1", role: "user"}` and `{id: "u2", role: "admin"}`).
   - A protected `DELETE /posts/:id` endpoint.
2. **Implement a custom middleware** `authorize(roles: string[])` that:
   - Reads the JWT from `Authorization: Bearer …`
   - Verifies it, extracts `role`
   - Blocks the request with `403` if `role` is not in `roles[]`, otherwise calls `next()`
3. **Deliverables:**
   - The Express app code (no frameworks beyond Express + jsonwebtoken).
   - **3 unit tests** (using Jest or Mocha) covering:
     1. Successful delete by admin
     2. Forbidden delete by normal user
     3. Missing/invalid token

---

## □ Part 3: React Front-end & Custom Hooks (≈ 1.5 hours)

> **Scenario:** Build a mini-feed UI that shows posts from the API above.

1. **Create a React SPA** (no CRA/Codesandbox—manual Webpack or Vite boilerplate is fine):
   - A login form that calls `/login`, stores JWT in memory (context or hook).
   - A feed page that fetches `/feed` & displays posts.
2. **Implement a custom hook** `useApi(resource: string, options?)` that:
   - Manages fetch, JSON-parsing, loading, error, and **caching** (so re-rendering the same resource doesn’t re-fetch).
3. **Add infinite scroll** to the feed (load 10 items at a time).
4. **Bonus (optional):** Optimize rendering so only new items mount/unmount, not the whole list.

---

## Part 4: Debugging & Code Review (≈ 30 minutes)

> **Scenario:** You've inherited this snippet. It's meant to fetch, sort, and return posts—but sometimes it returns duplicates or hangs.

```
// postsController.js
async function getSortedPosts(req, res) {
  const posts = await Posts.find();         // Mongoose model
  posts.sort((a, b) => b.created - a.created);
  res.json(posts);
}

// router.js
router.get('/posts', async (req, res) => {
  await getSortedPosts(req, res);
  console.log('Done.');
});
```

1. **Identify at least two problems** (bugs or performance issues) here.
2. **Propose corrections** (code snippets or descriptions).
3. **Explain** in plain English why each fix matters.

---

### Submission Checklist

- A **single Git repo** with four folders: `db/`, `api/`, `web/`, `debug/`.
- A top-level `README.md` listing:
  - Your schema designs and index notes (Part 1).
  - Instructions to run your API & UI.
  - Test commands & results.
- Keep dependencies minimal and versions explicit.
- Total work should take **no more than 4–5 hours** for a strong senior candidate.

---

**Scoring focus:**

- **Correctness & completeness**
- **Clarity of code & comments**
- **Test Coverage & edge-case handling**
- **Thoughtful schema/index choices**
- **Attention to performance & security details**

---

Good luck!