Cool, let's build a tiny Node + Express app and use it as a playground to **see flakiness** and then **fix it with Cypress**.

I'll go **step by step**, from project setup → app code → Cypress tests → how each topic is handled:

- Handling **dynamic elements & race conditions**
- Implementing **retries & timeouts**
- Debugging **flaky tests**

---

0. Project Setup

Step 0.1 – Create folder

mkdir cypress-flaky-demo
cd cypress-flaky-demo

Step 0.2 – Initialize Node project

npm init -y

Step 0.3 – Install dependencies

npm install express cors
npm install --save-dev nodemon cypress

Step 0.4 – Update `package.json` scripts

Open `package.json` and add these scripts:

```
"scripts": {
  "dev": "nodemon server.js",
  "start": "node server.js",
  "cypress:open": "cypress open",
  "cypress:run": "cypress run"
}
```

---

1. Build a Simple Express App

We'll have:

- `GET /` → serves `public/index.html`
- `GET /api/users` → returns users after **random delay** (to simulate dynamic UI)
- `GET /api/random-status` → sometimes 200, sometimes 500 (to simulate flaky behavior)

Step 1.1 – Create `server.js`

Create `server.js` in project root:

```
// server.js
const express = require('express');
const path = require('path');
const cors = require('cors');

const app = express();
const PORT = 3000;

app.use(cors());
app.use(express.json());

// Serve static files from public
app.use(express.static(path.join(__dirname, 'public')));

// Fake users data
const users = [
  { id: 1, name: 'Sachin' },
  { id: 2, name: 'Dravid' },
  { id: 3, name: 'Kohli' },
];

// GET /api/users with random delay (500–2500ms)
app.get('/api/users', (req, res) => {
  const delay = 500 + Math.floor(Math.random() * 2000); // 500–2500
  console.log(`[/api/users] Responding in ${delay}ms`);
```

```
  setTimeout(() => {
    res.json(users);
  }, delay);
});

// GET /api/random-status - sometimes success, sometimes failure
app.get('/api/random-status', (req, res) => {
  const isSuccess = Math.random() > 0.5; // 50% chance

  if (isSuccess) {
    console.log('[/api/random-status] 200 OK');
    res.json({ status: 'ok', message: 'Sometimes I pass' });
  } else {
    console.log('[/api/random-status] 500 ERROR');
    res.status(500).json({ status: 'error', message: 'Sometimes I fail' });
  }
});

// Serve index.html for /
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

Step 1.2 – Create frontend `public/index.html`

Create folder and file:

```
mkdir public
touch public/index.html
```

Put this in `public/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Cypress Flaky Demo</title>
```

```
  <style>
    body { font-family: Arial, sans-serif; padding: 20px; }
    #spinner { display: none; }
    .hidden { display: none; }
    .user-item { margin: 4px 0; }
    .error { color: red; }
  </style>
</head>
<body>
  <h1>Cypress Flaky Demo</h1>

  <!-- SECTION 1: Dynamic Users -->
  <section>
    <h2>Users (dynamic API)</h2>
    <button id="loadUsersBtn" data-testid="load-users-btn">Load Users</button>
    <div id="spinner" data-testid="spinner">Loading...</div>
    <ul id="userList" data-testid="user-list"></ul>
  </section>

  <hr />

  <!-- SECTION 2: Unstable API -->
  <section>
    <h2>Unstable API (random success/failure)</h2>
    <button id="checkStatusBtn" data-testid="check-status-btn">
      Check Random Status
    </button>
    <p id="statusMessage" data-testid="status-message"></p>
  </section>

  <script>
    const loadUsersBtn = document.getElementById('loadUsersBtn');
    const spinner = document.getElementById('spinner');
    const userList = document.getElementById('userList');

    const checkStatusBtn = document.getElementById('checkStatusBtn');
    const statusMessage = document.getElementById('statusMessage');

    // Load users with spinner
    loadUsersBtn.addEventListener('click', () => {
      userList.innerHTML = '';
      spinner.style.display = 'block';

      fetch('/api/users')
```

```
      .then((res) => res.json())
      .then((users) => {
        spinner.style.display = 'none';
        users.forEach((u) => {
          const li = document.createElement('li');
          li.textContent = u.name;
          li.classList.add('user-item');
          li.setAttribute('data-testid', 'user-item');
          userList.appendChild(li);
        });
      });
  });

  // Call unstable API
  checkStatusBtn.addEventListener('click', () => {
    statusMessage.textContent = 'Checking...';
    statusMessage.classList.remove('error');

    fetch('/api/random-status')
      .then((res) => {
        if (!res.ok) {
          throw new Error('Server error');
        }
        return res.json();
      })
      .then((data) => {
        statusMessage.textContent = data.message;
      })
      .catch(() => {
        statusMessage.textContent = 'Error from server';
        statusMessage.classList.add('error');
      });
  });
 </script>
</body>
</html>
```

Step 1.3 – Run the app

npm run dev

Open: `http://localhost:3000`

- Click **“Load Users”** → spinner → list appears after random delay.
- Click **“Check Random Status”** → sometimes success message, sometimes error.

This is perfect for demos of **dynamic UI** and **flakiness**.

---

2. Setup Cypress

Step 2.1 – Initialize Cypress

In a new terminal (keep server running):

npx cypress open

This will create `cypress` folder (or `cypress/e2e` in newer versions) and a basic `cypress.config.js` (or `.ts`).

Step 2.2 – Configure `baseUrl`

Open `cypress.config.js` and set:

```
// cypress.config.js
module.exports = {
  e2e: {
    baseUrl: 'http://localhost:3000',
    retries: {
      runMode: 2,
      openMode: 0,
    },
    defaultCommandTimeout: 8000, // 8s – more generous for slower UI
  },
};
```

---

3. Handling Dynamic Elements & Race Conditions

We’ll write two tests:

1. A **bad** (flaky) test.

2. A **fixed** (stable) test.

Step 3.1 – Create spec file

Create file: `cypress/e2e/users.cy.js` (or similar path Cypress uses):

```
// cypress/e2e/users.cy.js

describe('Users page - dynamic elements & race conditions', () => {
  // ❌ BAD / FLAKY VERSION
  it('naive test: loads users (flaky)', () => {
    cy.visit('/');

    cy.get('[data-testid="load-users-btn"]').click();

    // Immediately assert list length (no wait for API or spinner)
    cy.get('[data-testid="user-item"]').should('have.length', 3);
  });

  // ✅ GOOD / STABLE VERSION
  it('stable test: waits for API + spinner to finish', () => {
    cy.intercept('GET', '/api/users').as('getUsers');

    cy.visit('/');

    cy.get('[data-testid="load-users-btn"]').click();

    // 1. Wait for API to finish (no race condition)
    cy.wait('@getUsers');

    // 2. Ensure spinner disappears (UI settled)
    cy.get('[data-testid="spinner"]').should('not.be.visible');

    // 3. Now assert users list
    cy.get('[data-testid="user-item"]').should('have.length', 3);
  });
});
```

What's happening here?

- In **naive test**:
    - We click the button and *immediately* ask Cypress: “Are there 3 items?”
    - Because `/api/users` has a random delay, sometimes the users aren’t rendered yet → test fails → **race condition**.
- In **stable test**:
    - `cy.intercept('GET', '/api/users').as('getUsers');`
      Cypress watches this network call.
    - `cy.wait('@getUsers');`
      Cypress waits until the API call completes.
    - `cy.get('[data-testid="spinner"]').should('not.be.visible');`
      Wait until UI has removed spinner → DOM is stable.
    - Then assertions on the list.

This is a **textbook pattern** for:

> **Handling dynamic elements & race conditions using event-based waits instead of `cy.wait(5000)`**

---

4. Implementing Retries & Timeouts (Properly)

Cypress already retries commands like `cy.get` and `.should`.
Let’s use that and also show timeouts.

We’ll add another spec for the unstable API.

Step 4.1 – Flaky test for random-status API

Create file `cypress/e2e/random_status.cy.js`:

```
// cypress/e2e/random_status.cy.js

describe('Random status API - retries & timeouts', () => {
  // ❌ BAD: This test is truly flaky (depends on random backend behavior)
```

```
  it('naive test: sometimes passes, sometimes fails', () => {
    cy.visit('/');

    cy.get('[data-testid="check-status-btn"]').click();

    // We expect success message, but API may fail randomly
    cy.get('[data-testid="status-message"]').should('contain', 'Sometimes I pass');
  });
});
```

If you run this multiple times, sometimes it'll pass, sometimes fail → **true flaky test**.

Step 4.2 – Make it deterministic using `cy.intercept` (good pattern)

Let's fix it by stubbing API response:

```
// cypress/e2e/random_status.cy.js

describe('Random status API - retries & timeouts', () => {
  it('stable test: stub the API to always succeed', () => {
    // 1. Stub /api/random-status to always return success
    cy.intercept('GET', '/api/random-status', {
      statusCode: 200,
      body: { status: 'ok', message: 'Sometimes I pass' },
    }).as('randomStatus');

    cy.visit('/');

    cy.get('[data-testid="check-status-btn"]').click();

    // 2. Wait for stubbed call (ensures timing is handled)
    cy.wait('@randomStatus');

    // 3. Assert on message (Cypress retries this until timeout)
    cy.get('[data-testid="status-message"]', { timeout: 10000 }) // 10s
      .should('contain', 'Sometimes I pass');
  });
});
```

Where are retries & timeouts used here?

Global retries (test-level)

In `cypress.config.js` we set:

```
 retries: {
  runMode: 2,
  openMode: 0,
}
```

1.
    - In CI (`runMode`), Cypress will run a failing test up to **2 extra times**.
    - This doesn't *fix* logic bugs, but protects you from occasional environment issues.
2. **Command retry**
   `cy.get(...).should('contain', 'Sometimes I pass')`
    - Cypress automatically retries this until:
        - the condition passes OR
        - `{ timeout: 10000 }` ms passes (10 seconds here).
3. **Per-command timeout**
   We added `{ timeout: 10000 }` to `cy.get` – this overrides the global `defaultCommandTimeout` **only for this command**.

So:

- **Use `cy.intercept + cy.wait`** to avoid race conditions.
- **Use `.should(...)`** with optional **per-command timeout** to let Cypress retry for you.
- Only rely on **test retry (`retries`)** as a safety net, not as a main solution.

---

5. Debugging Flaky Tests – Systematic Workflow

Now imagine you start with the **naive flaky tests** above. How do you debug?

Let's walk through a concrete workflow using our app.

Step 5.1 – Catch a flaky failure

Run the naive tests:

npx cypress run --spec cypress/e2e/users.cy.js
npx cypress run --spec cypress/e2e/random_status.cy.js

1.
2. You'll notice:
    - `users.cy.js` sometimes fails when `/api/users` is slow.
    - `random_status.cy.js` fails randomly due to 500 responses.

Cypress will save:

- **Screenshots** on failure
- **Videos** (if enabled, default in `cypress run`)

Step 5.2 – Open in interactive mode to reproduce

npx cypress open

- Run only `users.cy.js`
- Run it multiple times
- When it fails, look at which command failed (e.g., `.should('have.length', 3)`).

Step 5.3 – Identify the root cause

Ask:

1. Is the assertion running **too early**?
    - For users: yes, API is delayed, DOM not ready.
2. Is the backend behavior **random**?
    - For random-status: yes, sometimes 500.

Step 5.4 – Use debugging helpers

You can inspect the DOM when the test fails:

```
cy.get('[data-testid="user-list"]')
  .debug()
  .find('[data-testid="user-item"]')
  .should('have.length', 3);
```

Or:

```
cy.get('[data-testid="user-item"]').then(($items) => {
  cy.log(`Found ${$items.length} items`);
});
```

In `cypress open`, you can add `debugger`:

```
cy.get('[data-testid="status-message"]').then(($msg) => {
  debugger; // Browser DevTools break here
});
```

Step 5.5 – Apply the correct fix

For **dynamic elements / timing** (users test):

- Add `cy.intercept` to watch the API.
- Use `cy.wait('@getUsers')`.
- Wait for spinner to disappear.
- Then assert on list → **fixed version** we already wrote.

For **random backend** (random-status test):

- Stub API with `cy.intercept` so test is deterministic.
- If you still want to test real API, keep that as a **separate test** or environment.

Step 5.6 – Confirm the fix

- Re-run tests many times in `cypress run`.
- Flaky ones should now be green consistently.
- If still flaky → collect failure info and repeat the cycle.

---

6. Recap – How This App Maps to Your Topics

✅ Handling dynamic elements & race conditions

- **Dynamic behavior:** `/api/users` responds with random delay.
- **Bad test:** immediately asserting list length.
- **Good pattern:**
  - `cy.intercept('GET', '/api/users').as('getUsers');`
  - `cy.wait('@getUsers');`
  - `cy.get('[data-testid="spinner"]').should('not.be.visible');`
  - Then assert on `[data-testid="user-item"]`.

✅ Implementing retries & timeouts

Global config:

```
 retries: { runMode: 2, openMode: 0 },
defaultCommandTimeout: 8000,
```

-

Per-command timeout:

```
 cy.get('[data-testid="status-message"]', { timeout: 10000 })
  .should('contain', 'Sometimes I pass');
```

-

- Understanding that `.should(...)` is **automatically retried** until it passes or times out.

✅ Debugging flaky tests

- Start with intentionally flaky tests:
    - Naive users test.
    - Naive random-status test.
- Use:
    - `cypress run` → screenshots/videos.
    - `cypress open` → reproduce & step through.
    - `cy.intercept`, `cy.wait`, `.debug()`, `cy.log`, `debugger`.
- Fix root cause:
    - Use event-based waits instead of fixed waits.
    - Stub random APIs to make tests deterministic.

If you want, next step I can:

- Add **one more test** that is flaky due to **shared state / test data** and show how to clean it up (DB or in-memory array).
- Or convert this into a **mini module** for your Cypress training slides: “Flaky test → root cause → fix pattern”.