Interviewer Assistance Questions for Bob Johnson

These questions test Bob's real-world problem-solving abilities based on his skills and gaps.

---

1. Selenium & C# - Debugging Flaky Tests

Scenario:

Bob is working on an automated test suite using **Selenium with C#**. Some tests pass locally but fail intermittently in the CI/CD pipeline. Logs show **timeouts and stale element exceptions**.

Question:

What steps would you take to debug and fix these flaky tests?

Expected Answer:

- Identify if test flakiness is due to **timing issues** (use explicit waits over implicit waits).
- Ensure elements are not **reloading asynchronously** (use `ExpectedConditions` for stable interactions).
- Investigate **DOM changes** (use robust locators like `XPath` instead of IDs that may change).
- Review **parallel execution** settings (ensure proper session handling in multi-threaded tests).
- Run tests **with different network conditions** to simulate real-world scenarios.
- Capture **screenshots and logs** on failure to analyze patterns.

---

2. CI/CD & GitHub Actions - Broken Deployment Pipeline

Scenario:

Bob set up a **GitHub Actions pipeline** to run tests and deploy applications. The deployment step fails intermittently, showing **permission errors and missing dependencies**.

Question:

How would you resolve these deployment issues?

Expected Answer:

- Check **GitHub Actions permissions** (ensure proper access tokens and permissions for deployment).
- Validate dependency installation (add `cache` actions to avoid missing dependencies).
- Ensure environment variables are correctly configured in **GitHub Secrets**.
- Review **workflow logs** to identify specific failure points.
- Test the pipeline **on a local runner** to replicate and debug the issue.
- Implement **retry logic** for transient failures in API or cloud deployments.

---

3. Security Testing - Identifying Vulnerabilities

Scenario:

A web application Bob is testing **stores sensitive customer data** but does not enforce **secure authentication mechanisms**. The app is vulnerable to **SQL injection and XSS attacks**.

Question:

What security testing strategies would you apply to identify and mitigate these vulnerabilities?

Expected Answer:

- Use **input validation and parameterized queries** to prevent SQL injection.
- Implement **content security policies (CSP)** to block malicious scripts.
- Perform **fuzz testing** to check for security loopholes in inputs.
- Run **static and dynamic security scans** (e.g., OWASP ZAP, Burp Suite).
- Test **session management security** (e.g., secure cookies, proper token expiration).
- Recommend using **multi-factor authentication (MFA)** to enhance security.

---

4. Azure DevOps - Test Automation Strategy

Scenario:

Bob's team is migrating test automation from a **local environment to Azure DevOps pipelines**. Test execution times have increased significantly, causing **delayed releases**.

Question:

How would you optimize test execution in Azure DevOps?

Expected Answer:

- **Parallelize test execution** using multiple agents in Azure DevOps.

- Use **test filtering** to run only relevant tests (e.g., smoke tests before full regression).
- Implement **caching strategies** to avoid redundant builds.
- Offload long-running tests to **cloud-based test grids** (e.g., Selenium Grid, Azure Test Plans).
- Optimize test scripts by **removing redundant steps** and improving wait strategies.
- Use **headless execution** in browsers to speed up test runs.

---

5. NoSQL & Terraform - Infrastructure Automation Challenge

Scenario:

Bob needs to deploy a **MongoDB NoSQL database** using **Terraform** for infrastructure automation. The deployment fails due to **incorrect IAM role configurations and missing state management**.

Question:

What steps would you take to successfully deploy and manage MongoDB with Terraform?

Expected Answer:

- Define proper **IAM roles and permissions** in Terraform for MongoDB deployment.
- Use **Terraform state management** (store state in **remote backend like S3** to avoid conflicts).
- Validate **Terraform configurations** using `terraform validate` before applying changes.
- Implement **module-based architecture** for reusable infrastructure code.
- Perform **dry runs** (`terraform plan`) before applying changes.
- Use **environment-specific variable files** to separate dev, staging, and production setups.