**Vijay Kumar Reddy**

* DOB – 10 AUG 1991
* GC ON 09 JUNE 2022
* 2012 June B. Tech
* 2012 July – 2015 Jan (CGI) – 2.5 Years
* Got Married in 2015 Feb
* Came to USA on 2015 March on H4 Visa
* 2015 June to 2016 Feb (H4 EAD Process)
* 2016 March to 2019 April (Progressive\_H4 EAD) Mayfield, OH – 3 Years
* 2019 May to 2020 Nov (Johnson & Johnso\_H4 EADBrunswick, New jersey – 1.5YR
* 2020 Dec to 2022 Jan (Target\_GCEAD) Minneapolis – 1 year
* 2022 Feb to Till Now (Huntington\_GC)- Columbus, Ohio– 2 years.

**INTRODUCTION:**

Hi, I’m Vijay. I’m a **Sr**.**QA engineer** with over 10+ years of experience in **manual** and **automation** testing for **web** and **mobile** applications using various tools and technologies. I have good experience in **Test Planning, Test Design, Test Environment, Test Execution, Defect Tracking, Test Reporting, and release sign off reports**. I have also experience in working with **agile** methodologies, **CI/CD** pipeline, and **Jenkins**. I have worked on several projects including **Banking E-commerce, Insurance, and HealthCare** for different clients. I used different testing tools to create and execute automated tests. I also have good **communication** skills to collaborate with the **development** team, the business team, and the offshore team.

I have good experience with**, API Testing**, **UI Testing**, **Database Testing**, **Performance Testing**, **Mobile Testing**.

**API Testing:**

**I have done different types of API Testing:**

**Unit Testing** to Focuses on individual API methods or functions.

**Functional Testing** to Verify the API’s functionality, including input validation, response validation, and error handling.

**Integration Testing** to Ensure that APIs work seamlessly together.

**Security Testing** to Check for vulnerabilities like SQL injection, authentication flaws, and authorization issues.

**Performance Testing** to EvaluateAPI response times, throughput, and scalability.

**Load Testing** to Measure how well an API handles concurrent requests.

**Stress Testing** to Push the API to its limits to identify breaking points.

**Tools used for API Testing:**

I used **Postman** for manual API testing to create and execute test cases, validate responses, and manage collections.

I have worked in Automating the API’s using **Rest Assured libraries in Java**. So, it simplifies sending HTTP requests, validating responses, and handling authentication. So, I can reuse the code and save execution time. I used **Playwright** Built in assertions to validate the API response, such as checking the **status code, response header and response body**. I used **TestNG** to integrate well with Rest Assured. So, it provides features like assertions, grouping, parallel execution, and reporting.

**UI Testing:** I used **Playwright** to work on a web automation framework across different browsers and platforms.

I have used **Playwright** to automate end to end testing of web applications and streamline testing process.

I have used **Automatic Waiting and Timeouts** to improve the stability and reliability of my test **automation scripts** and Implemented **Parallel Test Execution** to reduce the overall test execution time.

I have used **Playwright Inspector** to generate the Selectors and **Trace Viewer** to investigate the Test failure.

I have experience in BDD frameworks like **cucumber** which is built on top of **Selenium** for acceptance testing. I have good experience on **Browser Stack and Sauce Labs** for executing on different browsers like **Chrome, Safari,** **Edge**, and **Firefox** to Test manually. I have worked on Selenide which is a framework built on top of Selenium that simplifies the usage of Selenium. It provides a more concise API and better error messages.

**Database testing:** I have experience working in Relational databases **like SQL SERVER**, **MY SQL**, and **No SQL** databases like **COUCH DATABASE.**I used **Spring JDBC** for interacting with databases and ensuring data integrity during test execution. Am expertise in writing **simple** and **complex** queries to check the consistency and integrity of the database. I used JDBC Template to connect to the database and validate the data, to make sure the data is in the format. I also have best practices for database testing, such as creating test data, using test cases and scenarios, comparing expected and actual results, reporting defects and errors, and documenting the test results.

**Performance testing:** I have an experience in Performance testing using tools like **JMeter**, **Gatling.** I used to design and execute rigorous **performance tests** to assess system **performance**, application scalability, **reliability**, and **responsiveness** under various load conditions. In addition to designing and executing performance tests using tools like **JMeter** and **Gatling** enables testers to accurately assess system performance, scalability, and reliability under various load conditions. I have a deep understanding of performance testing methodologies and best practices. This includes analyzing **system performance** metrics, identifying **performance bottlenecks**, and recommending optimizations to enhance application scalability and reliability. Moreover, Iam expertise in **integrating performance testing** into **CI/CD** pipelines to enable continuous performance monitoring and ensure early detection of performance regressions.

**Mobile testing:** I have experience in working mobile application testing **both manual and automation**. I worked on mobile automation testing using **Appium.** I have tested applications across various devices, screen sizes, and operating systems, ensuring compatibility and consistency. I used Appium Desktop with the drivers and to connect the phone and I passed the desired capability. I used **XCUI** test for **IOS** testing and **UI Automator2** and for **android** testing. I am proficient in setting up and configuring **Appium** **Desktop** with drivers to establish connections between devices and testing environments, allowing for seamless automation testing execution.

**Roles and Responsibilities:**

* I used to **Develop** and **execute** the overall test strategy for the project.
* By using **Playwright** i have designed and developed a **test automation framework** for testing the **online Banking** applications.
* Creating and maintaining automated test scripts using **Playwright** for **functional**, **regression**, and **performance** testing, ensuring their reusability and maintainability to support ongoing development cycles.
* I have used **Playwright’s built in trace viewer** for debugging and troubleshooting.
* I will be allocating the **tasks** among the testing team members based on their skill sets and strengths by conducting regular team meetings to discuss progress, risks, and challenges**.**
* Identify test cases suitable for automation and **prioritize** them based on business impact.
* I oversee **defect management processes**, ensuring that defects are **tracked**, **prioritized**, and resolved in a timely manner. Additionally, I will be conducting the root **cause analysis (RCA)** to identify underlying issues.
* I will be Collaborating effectively with **cross-functional Agile teams**, including **developers**, **business** **users**, and other team members.
* documenting, and prioritizing defects using bug tracking tools like **JIRA** and collaborating with development teams to ensure timely resolution of issues.
* I Used to Integrate automated **test suites** into **CI/CD** pipelines for seamless and continuous testing. I will be executing the automated tests in the CI/CD pipeline to quickly check how well the application is working.
* I regularly provide updates and reports on testing progress, issues, and risks to project stakeholders. This involves effective communication with project managers, development teams, and business stakeholders to ensure alignment and transparency throughout the testing process.

**Sprint Planning**

I am part of an **18-member team** that includes **Project Manager**, **Business Analyst**, **Product Owner**, **developers**, and **testers**. Our sprint cycle is **2 weeks** long and we follow the **agile methodology.** We use **JIRA** for Task Management tool and **zephyr** for the Test Case Management Tool In the sprint planning meeting, I reviewed the user stories and acceptance criteria with the **BA** and the rest of the team. I will be estimating the size of each task and assign them to the appropriate **team** members During the sprint, I collaborate with the developers and other testers through **GitHub**, where we push and review our code. I also attend daily stand-up meetings to report my progress and challenges. I report and track defects found during testing using Jira platform. I participate in defect triage meetings (**Defect review board**) three times a week to prioritize and assign defects to developers. Before the demo session, I perform **regression testing** to ensure that all the features are working as expected. I use **Jenkins** to run my automated test cases and generate reports. I attended the demo session where we presented our work to the **PO in the UAT environment**. I provide feedback and suggestions for improvement. After the demo session, I attended the retrospective meeting where we discussed what went well and what can be improved in the next sprint. I also prepare for the production release by ensuring that all the defects are resolved, and all the test cases are passed.

**CI/CD Pipeline:**

* When Developers push the code to the **GITHUB** Repository. We use **Jenkins** as a Maven Build tool which will build the jar files.
* Jenkins polls with the GITHUB and invokes for any new code changes. where it does Smoke testing, Unit testing, Integration testing, as a tester I do regression testing.
* Jenkins will automate and execute test cases.
* We deploy code to the QA and UAT environment.

**Banking**: I have worked for Huntington Bank. I have good Experience in the payment’s domain and merchant processing.

I have worked with various banks, and payment devices. I tested various payment gateway vendors, like **Twilio**, **orbital,**

**pay** **flow**, **authorize.net** and **sage**. by calling their APIs and verifying the payment responses.

I have worked with different **Credit** **card** **transaction** types:

**Pre-authorization (Pre-auth)**: Reserving funds without completing the actual transaction.

**Authorization**: Approving a transaction.

**Capture**: Completing the transaction and transferring funds.

**Purchase (Sale)**: Actual purchase by the customer.

**Refund (Return)**: Reversing a transaction.

**Void**: Canceling an authorized transaction.

**Credit Card Chargeback**: Handling disputed transactions.

**performed Credit card validations:**

1. **Card Number Validation**: Making sure that the credit card number is correctly formatted and passes all the required validation checks.
2. **Expiration Date Validation**: Verifying that the expiration date provided is valid and in the future.
3. **CVV/CVC Validation**: Validating the Card Verification Value or Card Verification Code to match the card details and security measures.
4. **Address Verification System (AVS) Validation**: Comparing the billing address with the card issuer's records to confirm accuracy.
5. **Card Type Recognition**: Identifying the card type (e.g., Visa, Mastercard) based on the card number prefix.
6. **Risk Assessment**: Analyzing transaction patterns for potential fraud indicators or high-risk behaviors.
7. **Integration with Payment Gateway**: Sending credit card information securely to the payment processor for authorization and processing.
8. **Error Handling**: Managing validation failures and providing clear feedback to users for error correction.

In my role at Huntington Bank, I tested and implemented various **APIs** including.

* **Account Services** for retrieving account details and transactions,
* **Customer Information** for managing customer profiles,
* **Authentication** for secure user verification,
* **Transaction Categorization** for expense tracking,
* **Alerts and Notifications** for account alerts,
* **KYC** for identity verification,
* **Loan and Credit Score** for eligibility checks
* **Foreign Exchange** for real-time exchange rates, and Investment Management for portfolio access and transactions.

**(Second project in Huntington Bank)**

* And worked on testing **payment devices** like **Elavon lane** series and **Magtek** devices, by simulating different payment scenarios and checking the device outputs. We have a payment application where we take payments through different **payment mode.** Like **Card Payment, ACH Payment, Google Pay and Apple Pay**.
* In addition to testing payment devices and applications, I collaborated with cross-functional teams to ensure compatibility across different payment modes. This involved conducting integration tests to validate the seamless interaction between the **payment application** and various payment **gateways** or processors.
* Moreover, I also tested the integration of vendor pages with **apple pay** and **google pay** by passing cryptogram and device pan data and verifying the payment confirmation. Good experience in payments and payments plans where we offer monthly, biweekly plans based on rules engine to consumers.
* We also check for **duplicate payments** like for same account number for same amount with in one minute we reject the second payment as duplicate payment. We do store all our credit card information in our **vault**. Only automated process can decode when processing vault payments.
* We do handle **auth** and **capture workflow**. Our auth usually valid for 30days if not captured it will be voided. You can also void the auth that’s call reversal. We do handle **full and partial refunds**. I do have experience with payment device lane 10000 series where we can **Swipe DIP Tap** the card and make payments. This are considered as card present payment where the client gets better exchange rates from card brands.

-------------------------------------------------------------------------------------------

* Designed and documented detailed test cases to validate the functionality, security, and compliance of hardware payment devices with industry regulations and standards such as PCI-DSS.
* Conducted testing to verify EMV (Europay, Mastercard, Visa) compliance of hardware payment devices, including chip card transaction processing, terminal verification, and cryptographic verification.
* Tested the transaction processing capabilities of hardware payment devices, including authorization, settlement, and batch processing, to ensure accurate and timely transaction processing.
* Collaborated with software development teams to conduct integration testing of hardware payment devices with backend systems such as payment gateways, core banking systems, and fraud detection systems.
* Evaluated the usability and user experience of hardware payment devices, including keypad layout, display readability, and menu navigation, to ensure ease of use for bank staff and customers.
* Maintained detailed records of test results, defects, and corrective actions, and prepared comprehensive test reports and documentation for stakeholders, management, and regulatory compliance.

**Late fee grace**

The "late fee grace" feature provides users with an additional day to make a payment if they miss the payment due date. This grace period helps users avoid incurring late fees or penalties for minor delays in making their payments.

**Features:**

1. **Missed Payment:** If a user fails to make a payment on the specified due date, the system recognizes the missed payment.
2. **Grace Period Activation:** Upon detecting the missed payment, the system activates a grace period, typically lasting one day, during which the user can still make the payment without penalty.
3. **Additional Time to Pay:** During the grace period, the user has an extra day to submit the payment, effectively extending the deadline by one day beyond the original due date.
4. **Late Fee Avoidance:** If the user makes the payment within the grace period, no late fees or penalties are applied to their account.
5. **Notification:** Users may receive notifications or alerts informing them of the missed payment and the activation of the grace period. These notifications serve as reminders to make the payment promptly to avoid late fees.

Here are some scenarios to test the "**late fee grace**" feature and a scenario to explain it to the interviewer:

**Scenarios to Test the Feature:**

1. **Normal Payment Scenario:**
   * User makes a credit card payment on or before the due date.
   * Expected Result: No late fee is charged, and the payment is successfully processed.
2. **Missed Payment with Grace Period:**
   * User misses a credit card payment by one day.
   * Expected Result: The system grants a grace period, allowing the user an extra day to make the payment without incurring a late fee.
3. **Missed Payment Exceeding Grace Period:**
   * User misses a credit card payment by more than one day.
   * Expected Result: The system does not grant a grace period, and a late fee is charged to the user's account.
4. **Multiple Consecutive Missed Payments:**
   * User misses multiple consecutive credit card payments, each time within the grace period.
   * Expected Result: The system grants a grace period for each missed payment, but consecutive late payments may trigger additional warnings or penalties.
5. **Payment Made During Grace Period:**
   * User makes a payment within the grace period after missing the due date.
   * Expected Result: The payment is processed without incurring a late fee, and the account status is updated accordingly.
6. **Late Fee Calculation:**
   * User misses a payment and incurs a late fee.
   * Expected Result: The late fee is calculated accurately based on the credit card terms and conditions, considering factors such as the outstanding balance and the number of days overdue.
7. **Notification of Grace Period:**
   * User receives a notification or alert informing them of the grace period after missing a payment.
   * Expected Result: The notification clearly communicates the availability of the grace period and provides guidance on making the overdue payment.

**Challenges and Resolution**

**Issue:** Notification Delivery Reliability

**Challenge**: **Making sure users reliably receive notifications about missed payments and grace period** activation is a tricky task for QA Automation Engineers working in production

**Resolution Steps.**

==> Develop automated tests using **Playwright** to verify the accuracy and timeliness of notifications sent to users for missed payments and grace period activation.

==> Write Playwright scripts to test the system's error handling mechanisms, ensuring that notifications are queued and resent appropriately in case of delivery failures or errors

==> I have used Playwright’s capabilities to simulate diverse network conditions, such as poor connectivity or server downtime, to assess the resilience of the notification delivery mechanism.

Certainly, here are the steps we took to address the challenge of notification delivery reliability:

1. **Utilized Test Automation Tools**: Leveraged test automation frameworks like Selenium WebDriver and TestNG to develop comprehensive test suites covering various notification delivery scenarios. This allowed for thorough validation of the notification delivery process under different conditions.
2. **Employed API Testing Tools**: Utilized API testing tools such as Postman or RestAssured to validate the backend API endpoints responsible for sending notifications. This ensured proper communication between the application frontend and backend systems.
3. **Implemented Monitoring and Alerting Tools**: Deployed monitoring and alerting tools like Splunk or ELK Stack to continuously monitor the performance of the notification delivery process. These tools provided real-time insights into system behavior, enabling prompt identification and resolution of any issues or anomalies.
4. **Used Network Simulation Tools**: Employed network simulation tools like Apache JMeter or Gatling to simulate diverse network conditions during testing. By replicating scenarios such as poor connectivity or server downtime, we assessed the system's resilience and its ability to handle adverse network conditions.
5. **Leveraged Collaboration Platforms**: Utilized collaboration platforms such as JIRA or Microsoft Teams to facilitate communication and collaboration among team members. This ensured effective coordination during testing activities and streamlined the resolution of any identified issues.

By following these steps and leveraging these tools and technologies, we were able to effectively address the challenge of notification delivery reliability in our banking project. This resulted in an improved user experience and ensured compliance with regulatory standards.

1. **Complex Payment Flows**:

Payments involve processes such as fund transfers, bill payments, and payment gateways integration. Testing these flows comprehensively requires understanding various payment methods, currencies, regulations, and compliance standards.

* + **Resolution with Playwright**: Develop end-to-end test scripts using Playwright to simulate complex payment flows, covering various scenarios such as fund transfers, bill payments, and payment gateway interactions.
  + **Implementation**: Write test scripts to navigate through different pages of the banking application, inputting data, submitting forms, and verifying transaction outcomes. Utilize **Playwright's APIs** to interact with UI elements, handle pop-ups, and capture network requests/responses for validation.

1. **Transaction Security**:

Making sure money transactions are safe is really important. We must check how data is protected, how people log in securely, and if we follow the rules like **PCI-DSS** to keep card information safe.

* + **Resolution with Playwright**: Implement security-focused test scripts using Playwright to validate encryption protocols, secure authentication mechanisms, and compliance with standards like PCI-DSS.
  + **Implementation**: Develop test scripts to verify SSL/TLS encryption, validate secure authentication methods, and ensure compliance with security standards. Utilize Playwright's **network interception** capabilities to capture and analyze encrypted data exchanged during transactions.
  + Implemented multi-factor authentication (**MFA**) and tokenization techniques for enhanced user authentication and data protection.

1. **Integration Testing**:

Testing how banking systems work with other services like payment processors and banks can be tricky because they rely on those outside services. It's hard because we need to make sure they all work together smoothly, handle data correctly, and deal with problems well.

* + **Resolution with Playwright**: Perform integration testing using Playwright to validate interoperability, data consistency, and error handling between banking systems and third-party services.
  + **Implementation**: Write test scripts to interact with APIs of third-party services, simulate data exchange, and verify responses. Utilize Playwright's capabilities to mock external API responses, simulate network conditions, and validate data consistency across integrated systems. Integrate Playwright tests into CI/CD pipelines for automated regression testing and continuous validation of integrations.

**Test Automation Framework from Scratch Using Playwright and Typescript.**

**1. \*\*Initialize a new Node.js project\*\***

* + Firstly, I began by setting up the project structure, creating directories for test scripts, page objects, utilities, and configuration files.
  + Created a new directory and initialized a new Node.js project by running.

**npm init-y**

**2. \*\*Install dependencies\*\***

I have installed Playwright and TypeScript dependencies using npm to manage packages efficiently.

I installed different packages:

1. `**playwright**`: The Playwright library for automation
2. `@playwright/test`: Default Playwright Test Runner
3. `typescript`: The TypeScript compiler
4. `ts-node`: A TypeScript execution environment for Node.js
5. allure-playwright: Allure reporter integration for Playwright
6. @playwright/test-parallel: Parallel test execution for Playwright
7. @playwright/test-reporter: Custom reporters for Playwright
8. @faker-js/faker: Library for generating test data

* Created a `**tsconfig.json**` file with default TypeScript settings by initializing a new TypeScript project. This file configures the TypeScript compiler options.
* npx tsc --init
* In the page object model, we have maintained class for every web page which holds the functionality of web page.
* We have separate packages for pages and tests.
* In the page object model, we have maintained class for every web page which holds the functionality of web page.
* We have separate packages for **Pages** and **Tests**.
* Let’s say like for **homePage.ts, loginPage.ts, accountSummaryPage.ts** we have separate classes to store element locators.
* EX: a **LoginPage class** with methods like **navigateToLoginPage(), enterUsername(), enterPassword(), and clickLoginButton().**
* Created a new directory called **Tests** and add new files for your end-to-end tests and created **login.spec.ts, accountSummary.spec.ts,** etc.
* EX: **LoginTest class** with test methods like testSuccessfulLogin() and testInvalidCredentials().
* I Configured the Playwright Test Runner for that I have Created a new file **`playwright.config.ts`**
* we have separate configuration files for different environments (e.g., development, staging, production) or for storing sensitive data like credentials or API keys.
* This **configuration file** sets up the test runner options, including test retries, video recording, and support for cross-browser testing (Chromium, Firefox, WebKit, and Chrome mobile emulation). It also configures the Allure reporter for generating test reports.
* Update the `**package.json**` file, this file contains the project metadata, dependencies, and scripts for running tests. These scripts allow you to run the tests against all browsers or a specific browser, generate Allure reports, and open the Allure report.

For CI/CD integration, I have used GitHub Actions. GitHub Actions workflow file (ci.yml) for running tests, this workflow runs the automated tests on each pull request to the main branch, then run unit test and integration test and deployed in the targeted environment and smoke test will be performed there and allure reports.

**differences between the API and UI testing frameworks built from scratch using Playwright and TypeScript**

| **Aspect** | **API Testing Framework** | **UI Testing Framework** |
| --- | --- | --- |
| Dependencies | Axios for HTTP requests | Playwright for browser automation |
| Test Scripts | Interact with API endpoints and validate responses | Interact with web pages and validate UI elements |
| Test Execution | Run HTTP requests and verify responses | Execute in browser environment controlled by Playwright |
| Configuration | Set up Jest for test execution, configure Axios | Set up Jest with Playwright integration, configure Playwright |
| Test Environment | Headless execution without graphical interface | Requires graphical browser interface |
| Continuous Integration | Integration into CI/CD pipelines | Integration into CI/CD pipelines |

**How to Manage Test Data in Playwright**

* Use Test Data Files: Organize test data in JSON or CSV files and load them dynamically in tests.
* Generate Test Data Dynamically: Utilize libraries like Faker.js to generate random or synthetic test data within test scripts.
* Data-Driven Testing: Implement data-driven testing to iterate over datasets and run tests with different input values.
* Mock External Dependencies: Use mocking frameworks or Playwright's built-in mocking capabilities to simulate responses from external services.
* Database Seeding: Seed the database with test data before running tests to ensure consistent data for testing.
* Environment Variables: Pass sensitive or configurable data to tests using environment variables for secure management.
* Test Fixtures: Set up and tear down test data using fixtures before and after each test for a clean and consistent test environment.

**How to generate reports using Playwright.**

To generate reports in Playwright, you can use a library called "@playwright/test-reporter" to create Allure reports. You'll need to install this library using npm and configure it in your test runner setup file (playwright.config.ts). After configuration, run your tests with the --reporter option to generate the Allure reports.

**How to Perform parallel testing in Playwright**

* Install the "@playwright/test-parallel" package via npm.
* Configure your test runner file (e.g., playwright.config.ts) to enable parallel testing by setting the "parallel" option to true.
* Optionally, specify the number of parallel workers.
* Execute tests using the Playwright test runner with the "--parallel" option to run tests concurrently.
* Review the test results, which will include reports for each test run in parallel.

1. **how to integrate playwright with cicd and Jenkins?**

Sure, here's a summarized version of the steps to integrate Playwright with CI/CD and Jenkins in bullet points:

* **Install Playwright:**
  + Add Playwright as a dependency in your Node.js project using npm or yarn.
  + Include Playwright initialization in your test scripts.
* **Set up Jenkins:**
  + Install Jenkins on your server or use a Jenkins instance provided by a CI/CD service.
  + Create a new Jenkins job for your Playwright tests.
* **Configure Jenkins Pipeline:**
  + Add a pipeline script to your Jenkins job's configuration.
  + Define stages for installing dependencies, running tests, and publishing test results.
* **Run Playwright Tests in Jenkins:**
  + Jenkins will execute the pipeline steps, including installing dependencies with npm, running Playwright tests with **npx playwright test**, and publishing test results.
* **View Test Results:**
  + Jenkins displays test results, including pass/fail status and any errors encountered during test execution.
* **Optional: Integration with Reporting Tools:**
  + Integrate Jenkins with reporting tools like Allure or TestNG for detailed test reports and enhanced visibility into test results.

**which build tool is used for playwright?**

For Playwright, the recommended build tool is npm (Node Package Manager).

Playwright is typically installed and managed as a Node.js package using npm, and npm is also used for managing dependencies, scripts, and build configurations for Playwright-based projects.

**how to integrate playwright with browsers lab and scause labs?**

Sure, here's a human-readable summary of how to integrate Playwright with BrowserStack or Sauce Labs for cross-browser testing:

1. **Sign Up:** Create an account on BrowserStack or Sauce Labs if you don't have one already.
2. **Get Credentials:** Obtain your username and access key from the respective platform's dashboard.
3. **Install Packages:** Use npm to install the necessary packages for integrating Playwright with BrowserStack (**playwright-browserstack**) or Sauce Labs (**playwright-saucelabs**).
4. **Update Test Scripts:** Modify your Playwright test scripts to include the configuration for BrowserStack or Sauce Labs, providing your username and access key in the script.
5. **Run Tests:** Execute your Playwright tests as usual, and they will automatically run on BrowserStack or Sauce Labs using the specified browser and configuration.

By following these steps, you can seamlessly integrate Playwright with BrowserStack or Sauce Labs to conduct cross-browser testing and ensure your web applications work smoothly across different browsers and environments.

**E commerce Project: (Target)**

In my previous project, I was involved in testing an end-to-end e-commerce platform, where I played a pivotal role in ensuring the smooth functioning of various components, including the **Point of Sale (POS) system.** The eCommerce project revolves around the creation and for purchasing and selling goods or services. As part of my responsibilities, I ensure the **correct generation**, **alteration**, and **cancellation** of orders, guaranteeing that all order details are accurately captured in the system and reflect real-time changes. Moreover, I validate transactions, oversee **payment processing**, **manage orders**, and synchronize inventory seamlessly within the Point of Sale (**POS**) environment. This involves meticulous testing to confirm that all aspects of the POS system function smoothly and efficiently, providing a seamless shopping experience for users. I participated in developing and documenting user stories, outlining the test strategy, and estimating automation efforts.

Afterwards, I thoroughly examined and tested every **microservice**, covering areas such as **authentication,** **product catalog management**, **shopping cart functionality, order processing**, **payment gateway integration**, **inventory management**, user profile handling, recommendation engine, review and rating systems, notification services, analytics, and search capabilities.

Specifically, regarding the POS system, I tested its **transaction processing capabilities**, **payment integrations,** **loyalty program functionalities**, offline mode resilience, **integration with inventory management**, **handling of returns and exchanges**, **receipt generation**, compliance with regulatory standards, scalability, performance, usability, and accessibility. I ensured that the POS system seamlessly integrated with other components of the e-commerce platform and facilitated secure and efficient transactions between customers and the business.

Throughout the project, I collaborated closely with cross-functional teams, conducted code reviews to ensure the quality and reusability of test automation artifacts, and presented appropriate results to sign off on software functionality for new production releases.

Additionally, I actively participated in analyzing major incidents and identifying opportunities to automate software quality test cases to enhance efficiency and reduce risks.

Overall, my experience in testing the end-to-end e-commerce platform, including the comprehensive testing of the POS system, has equipped me with valuable insights and expertise in ensuring the reliability, performance, and usability of critical components in complex software applications.

**Performance testing in Banking project:**

* For performance testing, we used tools like **JMeter** to create test scripts that simulated concurrent user interactions with the application. We defined different **load profiles** based on expected user traffic and executed performance tests to measure **response times, transaction throughput**, and **server resource utilization.**
* JMeter provides various **listener**s and **reports** to monitor server response times, throughput, error rates, and other performance metrics during testing, enabling testers to analyze and optimize system performance.
* JMeter supports testing APIs by sending HTTP requests and analyzing responses, making it suitable for performance testing of RESTful and SOAP web services.
* By simulating user interactions and defining **load profiles**, performance testers can accurately replicate real-world scenarios, assessing key metrics like response times, transaction throughput, and server resource utilization.
* Scalability and stress testing help evaluate the application's ability to handle growing user demand and extreme load conditions, while transaction integrity testing ensures the accuracy and consistency of banking transactions. Continuous monitoring and analysis enable proactive detection of performance issues, facilitating optimization and maintenance of a seamless user experience. Overall, **JMeter** serves as a vital tool for proactively identifying and addressing performance bottlenecks, safeguarding the reliability and scalability of banking applications.

**JMeter vs Gatling**

| **Feature** | **JMeter** | **Gatling** |
| --- | --- | --- |
| Architecture | Thread-based | Asynchronous, event-driven |
| Scripting Language | XML-based, supports multiple languages | Scala-based DSL |
| Execution Model | Synchronous | Asynchronous, non-blocking |
| Reporting | Basic built-in reports, plugins available | Detailed HTML reports |
| Ease of Use | User-friendly GUI interface | Requires familiarity with Scala |

**Kafka in Real time.**

In a banking project, Kafka is used as a messaging system to handle real-time transaction data from various sources, such as ATM transactions, online banking activities, and mobile payments. As a QA Automation Engineer involved in testing the banking platform, your role includes ensuring the reliability, scalability, and accuracy of Kafka-based transaction processing.

**Health Care Project: (Johnson & Johnson)** In the Health Care Project mainly I will be doing **Usability and Accessibility Testing, Integration Testing.**

I have extensive experience in QA automation within the healthcare industry, focusing on testing electronic health record (EHR) systems, health information exchange (HIE) platforms, **telemedicine** solutions, and regulatory compliance tools. My responsibilities included developing and executing automated test scripts to validate critical functionalities, ensuring seamless communication, reliable data transmission, and compliance with regulatory standards such as HIPAA. I'm confident in my ability to leverage automation to improve efficiency, quality, and patient outcomes in healthcare projects."

**"**In my previous role in healthcare, I collaborated closely with the Clinical Software application Product, UX, and Engineering teams within our assigned pod. We worked together to ensure that our software solutions met the needs of our users and adhered to industry standards and regulations.

One of my key responsibilities was reviewing and analyzing business and system requirements or specifications. This involved meticulously examining the details to ensure a thorough understanding of the project scope and objectives. By doing so, I was able to identify potential areas for improvement and contribute valuable insights to the development process.

As part of my role, I was also tasked with designing comprehensive test plans, scenarios, test cases, and scripts. These documents served as the blueprint for our testing efforts, outlining the steps to be taken and the expected outcomes. Through careful planning and attention to detail, I ensured that our testing strategy aligned with the project goals and requirements.

When it came to execution, I executed test cases, whether manual or automated, with precision and thoroughness. I meticulously logged test results and defects, providing clear and detailed documentation for our development and product teams. This facilitated efficient bug fixing and helped us identify any requirement updates or clarifications needed.

Moreover, recognizing the importance of efficiency and scalability, I worked to automate tests wherever possible. By leveraging existing frameworks and tools, I automated regression tests to streamline our testing process and minimize manual effort. This not only saved time but also improved the reliability and repeatability of our testing efforts.

Overall, my experience in healthcare has equipped me with the skills and expertise needed to effectively collaborate with cross-functional teams, analyze requirements, design test strategies, execute tests, and automate where feasible. I am confident that these abilities will enable me to contribute positively to your team and help deliver high-quality software solutions."

**Insurance Project: (Progressive)**

* In this I will be doing the **Security** **Testing**, **Integration** **Testing**, **Claims** **Processing** Testing, **Policy** **Administration** **System** Testing.
* In the **functional testing** Develop and execute test cases to validate the functionality of insurance policy management systems, claims processing modules, and customer portals.
* In the **Integration Testing** Design and execute test cases to verify seamless communication and data exchange between insurance system components, including policy management, CRM, billing, and external databases.
* In the **Regression Testing** Maintain and execute regression test suites to validate existing functionalities and identify regression issues caused by software updates or configuration changes.
* my role involves ensuring the **quality** and **reliability** of insurance software through a comprehensive testing approach.
* This includes **functional** testing to verify **key** functionalities such as **policy** management and **claims** processing, integration testing to ensure seamless communication between various system components, and **regression** testing to maintain stability during software updates.
* Additionally, I conduct **performance** testing to assess system scalability, security testing to identify and mitigate potential vulnerabilities, and user acceptance testing to ensure alignment with business requirements.

**Using Appium from Scratch in recent Project:**

* In our latest project, we used Appium mainly for testing mobile applications on both Android and iOS platforms.
* We began by setting up the Appium environment, installing the server using **npm**, and configuring it to listen on specific IP addresses and ports.
* Our team created virtual devices and emulators using Android Studio for Android and XCode for iOS, ensuring coverage across different mobile platforms.
* Once the application under test (AUT) was installed on the emulators and simulators, we started the Appium server and wrote test scripts using our preferred programming language.
* These scripts allowed us to interact with the mobile application's UI elements using selectors such as XPath and ID.
* During test execution, we monitored progress, debugged any failures, and analyzed test results to identify issues or defects in the application.
* Additionally, we integrated Appium tests with our continuous integration (CI) system to automate test execution as part of our CI/CD pipeline, enabling rapid feedback and ensuring the quality and reliability of our mobile software products.

**Page Factory:**

* Page Factory in Selenium is an inbuilt Page Object Model framework concept for Selenium WebDriver, but it is very optimized. It is used for initialization of Page objects or to instantiate the Page object itself. It is also used to initialize Page class elements without using “FindElement/s.”
* The benefits of using **Page Factory** include reduced code duplication, improved code readability, and improved test maintenance. By creating Page Objects with Page Factory, the automation developer can focus on the logic of the tests rather than the implementation details of the web elements.
* It uses a **FindBy** annotation for page objects. Each object doesn’t require initialization like POM, and all elements are initialized with **initElements().** It doesn’t have cache storage because it doesn’t need it.

**Database Testing in Banking Project**

In our banking project, we extensively utilized database testing to ensure the integrity, security, and performance of critical financial data stored within the system. Through data verification tests, we meticulously validated the accuracy of customer account balances, transaction records, and other essential financial data, ensuring alignment with expected values and business rules. Additionally, we implemented data integrity checks to maintain referential integrity and enforce data relationships within the database, safeguarding against data inconsistencies. Schema validation was integral to our testing process, ensuring that the database schema remained consistent across different environments and accurately reflected application requirements. Performance testing allowed us to assess database responsiveness under varying load conditions, identifying and addressing performance bottlenecks to optimize query execution times and throughput. Security testing was paramount to our efforts, ensuring that the database was fortified against unauthorized access, SQL injection attacks, and data breaches, with robust access controls and encryption mechanisms in place. Moreover, backup and recovery testing validated our disaster recovery preparedness, ensuring swift data restoration and minimal downtime in the event of system failures. Concurrency testing enabled us to assess the database's ability to handle multiple simultaneous transactions while maintaining data consistency and isolation levels. Lastly, ensuring cross-platform compatibility across different database management systems ensured consistent functionality and behavior, further enhancing the reliability and robustness of our banking application.

**Create an effective bug ticket?**

* Description: A summary of the bug. It should provide a clear picture.
* Priority: Indicates the urgency to fix the bug (e.g., low, medium, high, critical).
* Severity: Indicates the impact of the bug on the system (e.g., minor, major, critical).
* A step-by-step guide to reproduce the bug. This should be detailed enough that someone unfamiliar with the issue can replicate the problem.
* Expected Result
* Actual Result
* Screenshots or screen recordings can be very helpful.
* Logs or error messages that might help in diagnosing the issue.
* Specify the environment in which the bug was observed.
* Mention details like browser version, OS, device type (for mobile issues)

**Bug Life Cycle:**

New: When a new defect is logged and posted for the first time. It is assigned a status as NEW.

Assigned: Once the bug is posted by the tester, the lead of the tester approves the bug and assigns the bug to the developer team

Open: The developer starts analyzing and works on the defect fix

Fixed: When a developer makes a necessary code change and verifies the change, he or she can make bug status as “Fixed.”

Pending retest: Once the defect is fixed the developer gives a particular code for retesting the code to the tester. Since the software testing remains pending from the testers end, the status assigned is “pending retest.”

Retest: Tester does the retesting of the code at this stage to check whether the defect is fixed by the developer or not and changes the status to “Re-test.”

Verified: The tester re-tests the bug after it got fixed by the developer. If there is no bug detected in the software, then the bug is fixed, and the status assigned is “verified.”

Reopen: If the bug persists even after the developer has fixed the bug, the tester changes the status to “reopened”. Once again, the bug goes through the life cycle.

Closed: If the bug is no longer exists then tester assigns the status “Closed.”

Duplicate: If the defect is repeated twice or the defect corresponds to the same concept of the bug, the status is changed to “duplicate.”

Rejected: If the developer feels the defect is not a genuine defect, then it changes the defect to “rejected.”

Deferred: If the present bug is not of a prime priority and if it is expected to get fixed in the next release, then status “Deferred” is assigned to such bugs

Not a bug: If it does not affect the functionality of the application then the status assigned to a bug is “Not a bug”.

**Healthcare**

In my previous role, I was responsible for the quality assurance of the company's **healthcare claims processing system**. I worked closely with **developers** and other **QA** analysts to identify and **fix defects in the system**. I also developed and executed test plans to ensure that the system met all the **requirements**. I do have experience in the healthcare domain, mainly in insurance claims, benefits, and enrolment. I have a strong understanding of EDI 837 transactions as per HIPAA based transactional 5010 formats and in analyzing test EDI 837I and 837P files and modifying them appropriately to create different test data scenarios to validate various business rules of the application. I am also experienced in working with 999 (Acknowledgement), 277CA (Accepted with errors), 278 (Authorization Request and Response), 835 (Claim approval with EOB) type EDI transactions. I have a good understanding of eligibility enquiry (270) and response (271) type EDI transactions. I have worked with edit codes to validate HIPAA compliance of the application. I am also experienced in working with ICD10 and HIPAA (5010).

HIPAA-based Transactional 5010 Formats:

HIPAA mandates the use of specific EDI transaction standards, and the 5010 format is the current version that organizations must adhere to. It defines the structure and data elements for various healthcare transactions, ensuring consistent and accurate data exchange.

In the Health care domain, I have been responsible for creating comprehensive test plans, test cases, and test scripts to thoroughly validate the functionality of various applications, such as electronic health record (EHR) systems, patient management systems, and medical billing software. I have experience in performing functional, integration, regression, and user acceptance testing, often working closely with domain experts to ensure the software meets the specific needs of health care professionals.

In one project, we were dealing with a complex patient data synchronization process across multiple systems. To address this challenge, I developed a data-driven testing approach using Selenium and Excel spreadsheets.

Ensuring the security and privacy of patient information is of paramount importance in the health care domain.

Regularly reviewing and updating our testing procedures and ensuring compliance with industry regulations like HIPAA are also key aspects of maintaining data security.

I have good Experience in writing complex SQL queries to extract, transform, and analyze data from large and intricate health care databases. For instance, I have developed queries to retrieve patient records, medical history, treatment plans, and billing information. These queries often involve multiple joins, subqueries, and aggregate functions to provide comprehensive insights to medical practitioners and administrators.

**Health Care Domain Testing**

During my time at XYZ Healthcare Solutions, I played a key role in testing an electronic health record (**EHR**) system. My responsibilities included ensuring that all features of the **EHR** system, such as **patient registration** and **appointment scheduling**, functioned correctly according to specifications. I also collaborated with **integration teams** to ensure seamless interoperability with other healthcare systems. Additionally, I conducted regulatory compliance testing to ensure adherence to **healthcare** regulations like HIPAA and HL7. I evaluated the user interface design through usability testing sessions and assessed the system's performance and scalability under various load conditions. Security testing was also a priority to identify and address potential vulnerabilities. Finally, I facilitated user acceptance testing sessions with healthcare professionals to ensure that the **EHR** system met their needs and supported their workflows effectively. Overall, my experience in **healthcare testing** has provided me with valuable insights into the challenges and considerations of ensuring the quality and reliability of software systems in critical healthcare settings, ultimately contributing to improved patient care outcomes and healthcare delivery efficiency.

**Automation framework from scratch using selenium, cucumber, TestNG.**

* In our project we are using Data driven framework by using page object model design pattern and page factory. Initializing the web driver and used different types of locators like Id, name, Xpath, link text, partial link text, CSS selectors and used page object model.
* In page object model, we have maintained class for every web page which holds the functionality of web page.
* We have separate packages for pages and tests.
* Let’s say like for home page and login page we have separate classes to store element locators. For login test there would be a separate class which calls the methods from home page and login page.
* We used **maven** as build tool all the tests are kept in the **'src/test/java'** and remaining files such as config. properties, element locators, test data, etc., kept under **'src/main/java'**.
* **Test Base Class** is responsible for loading the configurations from properties files, Initializing the WebDriver, Extent Reports.
* **Utility class** stores and handles the functions which can be used across the entire framework. The main purpose of creating the utility class is to achieve reusability.
* **The Property files** are used to store the information that remains static throughout the framework such as browser-specific information, application URL, screenshots path, etc.
* **Screenshots** will be captured and stored in a separate folder and the screenshots of failed test cases will be added to the extent reports.
* All the historical test data will be kept in an excel sheet. By using test data xlsx', we pass test data and handle data-driven testing. We use Apache POI to handle excel sheets.
* Used TestNG for Parallel execution. Assertions, grouping.
* We use **cucumber** as BDD framework.
* Create a **feature file** in src/test/resources.
* Create the **Step Definition** class or Glue Code in src/test/java.
* Create a **TestNG Cucumber Runner class** in src/test/java.
* Run the tests from **TestNG Runner.**
* Run the tests from **TestNG.xmI**
* Used Maven for build, execution, and dependency purpose. Integrating the Test dependency in the POM.XmI file and running this POM.xml file using Jenkins.
* When the developer commits the code in git remote repository, the Jenkins will detect it and trigger the CI/CD pipeline. in the CI/CD pipeline first step is build the code, then run unit test and integration test and deployed in the targeted environment and smoke test will be performed there and run the automated tests and allure reports.

**What is cucumber key words?**

**Feature**: Each feature file begins with 'Feature' keyword. Every feature file should have a feature extension. It gives a summary of what you will be testing.

**Scenario**: Each feature file contains one or more multiple scenarios. Each test is called a scenario followed by three parts.

**GIVEN**: Defines precondition for the test. For example, if you want to verify that a company logo is displaying, the precondition to verify the logo is displaying would be that the user is at the Home Page.

**WHEN**: Defines that an action is performed by the user.

**THEN**: Defines the outcome of the previous steps.

**AND**: If you have multiple 'WHEN,' you can use 'AND.'

**Background**: you'll find yourself repeating the same Given steps in all of the scenarios in a Feature. A Background allows you to add some context to the scenarios that follow it. It can contain one or more Given steps, which are run before each scenario, but after any Before hooks.

Hooks: Hooks are blocks of code that can run at various points in the Cucumber execution cycle. They are typically used for setup and teardown of the environment before and after each scenario.

Scenario Outline:-The Scenario Outline keyword can be used to run the same Scenario multiple times, with different combinations of values. A Scenario Outline must contain One or more Examples. the scenario Outline is run once for each row in the Examples section beneath it (not counting the first header row).

The steps can use < > delimited parameters that reference headers in the examples table. Cucumber will replace these parameters with values from the table before it tries to match the step against a step definition.

Scenario Outline: eating

**Cucumber with selenium Connection**

First, we dependencies

We create Junit/ TestNG runner class.

Use @Runwith annotation. I specify (**Cucumber. Class)** is a parameter.

@cucumber. options I specify (feature is a parameter that provides path of feature file).

**Glue** is another parameter is a path of the step definition class. Plugin generates HTML reports on the provided locations.

**Best Practices writing cucumber Scripts?**

* Each Scenario Should Execute Separately.
* Every feature should be able to be executed along.
* Steps information should be shown independently.
* Connect your scenarios with requirements.
* Keep a complete track of what scenarios should be included in a required document.
* Create modular and easy to understand steps.
* Try to combine all your common scenarios.

***Common UI test cases?***

*Ideally, we can group UI elements into 3 major categories.*

***Input elements*** *– Input elements are responsible for handling different user inputs.*

* *Checkboxes*
* *Dropdowns*
* *Combo boxes*
* *Buttons*
* *Toggles*
* *Text/password fields*
* *Date pickers*
* *Checkboxes*
* *Radio buttons*
* *Confirmation dialogues*

***Output elements*** *– Output elements are responsible for showing results against various user inputs. They also show alerts, warnings, success, and error messages to the users.*

***Helper elements*** *– further divided into navigational, informational, and containers, they help to move through the digital product, get information, and point user’s attention to some element.*

* *Notifications*
* *Breadcrumbs*
* *Icons*
* *Sliders*
* *Notifications*
* *Progress bars*
* *Tooltips*

**Required Fields** - If the screen requires data entry on a specific field, it is good practice to identify the required fields with a red asterisk and to give a friendly warning if the data is left blank.

**Data Type Errors** - If the screen contains dates, numeric, currency or other specific data types, ensure that only valid data can be entered.

**Field Widths** - If the screen contains text boxes that allow data entry, ensure that the width of data entered does not exceed the width of the table field (e.g., a title that allows 100 characters in the database should not allow more than 100 characters to be entered from the user interface).

**Navigational elements** – Verify all navigational buttons on the page are working correctly, and that they redirect users to the right page or screen.

**Progress Bars** - If the screen takes more than 5 seconds to render results, it should contain a progress bar so that the user understands the processing is continuing.

Cosmetic Inconsistencies - The screen look, feel and design should match the other screens in your application. Creating and using a style guide is a great way to ensure consistency throughout your application.

**Save Confirmations** - If your screen allows changing of data without saving, it should prompt you to save if you move to another record or screen.

**Delete Confirmations** - If a user deletes an item, it is a good idea to confirm the delete. However, if your user interface allows deleting several records in a row, in some cases you might consider allowing them to ignore the confirmation as it might get frustrating to click the confirmation repeatedly.

**Error Messages** - Ensure that error messages are informative, grammatically correct, and not condescending.

**Common Cross browser testing UI test cases?**

**Cross Browser Testing** plays a crucial role in website development by verifying that your web application functions optimally across different browsers, operating systems, devices,

**Testing Browser-OS Combinations**:

* Verify compatibility on popular browsers: Firefox, Chrome, Edge, Safari, etc.
* Assess performance on different operating systems: Windows, macOS, iOS, Android, etc.
* Ensure consistent functionality and visual display across various browser-OS combinations.

**Testing on Different Devices:**

* Assess website responsiveness and usability on smartphones, tablets, desktops, laptops, etc.
* Verify that the layout adapts seamlessly to different screen sizes and resolutions.
* Test touch interactions, gestures, and device-specific functionalities.

**Challenges in UI Testing?**

* **Continuous upgrades**: Upgrading constantly to accommodate new features and functionalities means performing comprehensive UI tests is challenging.
* **Increasing complexity**: Modern applications have highly complex features such as embedded frames etc.
* **Cross Browser Adaptability:** This helps ensure that their web apps are easily accessible by users across multiple browsers.
* **Inconsistencies in the Page Layout across Devices**: Web applications tend to suffer from rendering inconsistencies when accessed through devices with different screen sizes. This ensures that the web Validate the API keys for minimum and maximum range/length.
* Verify correct HTTP status code.
* Identify and verify the handling of API error codes.
* Verify response payload Check the JSON Schema validation, the Field Type, and the Mandatory Fields
* Validate the Response headers key and values.

Test request chaining pp is responsive across multiple devices like tablets, smartphones, desktops, etc.

***Common API test cases?***

**Functional Test Cases:**

* Validate the API keys for minimum and maximum range/length.
* Verify correct HTTP status code.
* Identify and verify the handling of API error codes.
* Validate the response headers Key and Values.
* to identify the working of multiple APIs together
* Validate end-to-end CRUD (create, read, update, and delete) flow for application API.
* Check database integrity test cases.
* Verify file upload and download test cases.

**Non-Functional Test Cases:**

* Check that the API is designed according to correct security principles using different authentication mechanisms.
* Role Permissions: ensure that specific endpoints are exposed to the user based on role.
* Check API performance with response time, latency, TTFB/TTLB in various scenarios (in isolation and under load)

Elapsed: TTLB - time to last byte

Latency: TTFB - time to first byte

**Challenges in API Testing?**

* **API Parameter Combinations:**

APIs handle communication between systems by assigning data values to parameters and passing those parameters through data requests. It’s necessary to test all possible parameter request combinations in the API to test for problems pertaining to specific configurations. Adding an additional parameter exponentially increases the number of possible combinations.

* **API Call Sequences:**

When calling an API, a client application sends multiple requests, which must be called in the correct order. If the requests are handled in the wrong order, the program will return an error. An example of this would be the error that comes up when an API call to delete an object is made before the call to create it.

* **Validating Parameters:**

Validating to make sure all parameter data uses the correct string or numerical data type, fits within length restrictions, fits within a designated value range, and passes other validation criteria. For example, U.S. phone numbers should appear in a 10-digit format and returning a 5-digit zip code should trigger an invalidation error.

* **Updating the Schema of API Testing:**

The schema–in other words, the data formatting that handles requests and responses for the API–needs to be maintained throughout the testing process. Any updates to the program that create additional parameters for the API calls need to be reflected in the schema configuration.

**Types of Bugs that API testing detects**

* Fails to handle error conditions gracefully
* Timeout Issues
* Unused flags
* Missing or duplicate functionality
* Reliability Issues. Difficulty in connecting and getting a response from API.
* Security Issues.
* Multi-threading issues.
* Performance Issues. API response time is very high.
* Improper errors/warning to a caller.
* Incorrect handling of valid argument values.
* Response Data is not structured correctly (JSON or XML).

**How to decide what test cases to be automated?**

* If the feature operates on a critical path or is frequently used, it must perform the best. And automating their tests will help in achieving that.
* Regression tests these tests are consuming in terms of time and resources as they’re the backbone of each release’s testing process.
* Data-driven tests or Tests with several combinations: Test cases that involve different combinations or configurations are automated to save time and to avoid human errors.
* Performance tests (load test, stress test, etc.): they are repetitive and time-consuming to reach the desired coverage.

what is CAPA process in QA Testing?

We follow **Corrective Action and Preventive action Process**.

Even though we might do through regression. bugs can happen. CAPA is the process to eliminate those from happening.

**Corrective Action** starts with locating the root cause of the problem, and then taking necessary actions to eliminate the root cause of the problem. It is performed after the occurrence of a defect. We deploy a hotfix on production to fix a production incident caused by the most recent production deployment.

**Preventive Action** is an activity designed to prevent a future (or potential) defect from occurring. Intention here is to stop this problem from occurring in the future.

* Looking for how we missed this in the acceptance criteria. Look at RTM and see how it got missed.
* Root cause analysis process
* Retrospective process
* Reviewing organizational process assets like historical lessons learned, etc.

How to parse Json in Java

1)using Json object and JsoNArray (Java Standard library)

2)using Jackson Library

3)It provides objectmapper class that can map Json to Java objects.

4)using Json library using Json-processing) API Json Path and POJOS.

Created a TEST file under src/test/java to write the test code.

and did Test Execution through TestNG-Runner class and right-click Run as TestNG Test. The tests will run as TestNG.

tests

Run the tests from TestNG.xml-Create a TestNG.ml and run the tests as TestNG.

After the test execution, refresh the project, and a new folder with the name test-output will be generated. This folder contains the reports generated by TestNG.

**selenium web driver Architecture**

* Import the WebDriver library for your programming language.
* Choose a browser and its corresponding WebDriver implementation.
* Write test scripts using WebDriver API methods to automate browser interactions.
* Run the test scripts using a testing framework or a test runner.
* WebDriver sends commands to the browser driver, which in turn controls the browser to perform the desired actions.
* The browser driver communicates with the browser using browser-specific automation mechanisms.
* The browser performs the actions and sends back the results to the WebDriver driver.
* Your test script receives the results and can make assertions or validations based on them.

Selenium Error Debugging skill.

I use breakpoints, screenshots, and session logging to debug my code and identify the root cause of the bug.

**Breakpoints**: Breakpoints are points in the code where you want to pause the execution and inspect the variables, expressions, or outputs. You can use breakpoints to run your code step by step and see how it behaves at each line.

**Screenshots:** Screenshots are images that capture the state of the web page or the browser at a certain moment. You can use screenshots to verify the visual appearance of your web application, compare the expected and actual results, or troubleshoot any UI issues.

**Session Logging**: Session logging is the process of recording the events and actions that occur during a test session. You can use session logging to track the progress and status of your test execution, analyze the performance and errors of your web application, or reproduce any issues or failures.

4.Here's a simplified explanation of testing Android and iOS using Appium:

**1.Install Appium:**

Install Appium on your computer. It's a tool that helps automate testing for mobile apps.

**2. Setup Dependencies:**

Make sure you have the necessary software installed, like Node.js, Java (for Android), and Xcode (for iOS).

**3.Start Appium Server:**

Run the Appium server. It's like a traffic cop that helps communication between your computer and the mobile devices.

4. **Define Device Settings:**

Tell Appium about the mobile device you want to test (name, version, etc.). These are called "desired capabilities."

**5.Write Test Scripts:**

Use a programming language (like Java or Python) to write simple scripts that describe what actions the app should take during testing.

**6.Identify App Elements:**

Specify how to find and interact with elements in your app, like buttons or text fields.

**7.Run Your Scripts:**

Run your scripts. Appium translates your commands into actions on the mobile device, like tapping buttons or entering text.

**8.Check Results:**

See if your app behaves as expected. Appium provides logs and reports to help you understand what happened during the tests.

**9.Repeat for iOS:**

If you want to test on an iOS device, you follow a similar process, but you might need a Mac and Xcode installed.

**10.Make Testing Easier:**

Use tools like Appium Inspector to visually identify app elements, making it easier to write your scripts.

**11.Automate Everything (Optional):**

If you like, you can set up things, so your tests run automatically whenever you make changes to your app.

In short, Appium helps you automate testing for both Android and iOS by allowing you to write scripts that simulate user interactions on mobile devices, helping ensure your app works as expected.

**Playwright:**

* Playwright is a framework for Web Testing and Automation. It allows testing Chromium, Firefox and WebKit with a single API. Playwright is built to enable cross-browser web automation that is ever-green, capable, reliable, and fast. Headless execution is supported for all browsers on all platforms.
* **The primary goal of Playwright is improving automated UI testing.**
* Playwright comes with e integration. For example, the Playwright has Docker images, allowing you to run tests quickly in an isolated and controlled environment.
* They also support your existing JavaScript test runners, like Jest/Jasmine, AVA, and Mocha, which is helpful if you are porting from an existing code base.
* The framework supports cross-browser development on Chromium, Web Kit, and Firefox – including Chrome, Edge, Firefox, Opera, and Safari.
* Cross-platform execution is supported on Windows, Linux, and macOS.
* Testing cross-language, including JavaScript, TypeScript, Python, Java, and .NET – choose the environment that suits you while still covering all areas and formats.
* Auto-wait, smart assertions that retry until an element is found, and test data tracing – keep track of logs and videos easily.
* Built with modern architecture and no restrictions, the application allows you to interact with multi-page, multi-tab websites like a real user and easily handles frames and browser events.
* As the Playwright framework is aligned with the modern browser's architecture, it doesn't have the in-process test runner limitations.
* Playwright delivers full test isolation with no overhead. It creates a browser context for each test which only takes a handful of milliseconds.

**Cypress:**

* I would choose Cypress for its simplicity and fast execution, especially for non-Angular applications.
* Data-driven testing in Cypress involves iterating through test data sets.
* It's beneficial for testing different scenarios without duplicating test code. Data-driven tests enhance test coverage and make scripts more maintainable.
* In my previous role, I extensively used Cypress for end-to-end testing due to its ease of use and robust capabilities. I also have experience with Protractor for Angular applications, Zest for API testing, Enzyme for React component testing, and Mocha for JavaScript unit testing.

**Applying oops concept in Selenium Automation?**

**Page object Design Pattern**

 In Selenium, we call objects as locators (such as ID, Name, Class Name, Tag Name, Link Text, Partial Link Text, XPath, and CSS**). Object repository is a collection of objects**. One of the ways to create Object Repository is to place all the locators in a separate file (i.e., properties or a Page Object file). But the best way is to use Page Object Model. **In the Page Object Model Design Pattern, each web page is represented as a class. All the objects related to a particular page of a web application are stored in a class.**

**Why is the Main method static in Java?**

The main method is always static because static members are those methods that belong to the classes, not to an individual object. So, if the main method will not be static then for every object, it is available. And that is not acceptable by JVM.JVM calls the main method based on the class name itself. Not by creating the object. Because there must be only 1 main method in the java program as the Execution starts from the main Method. So, for this reason the main method is static.

1. **ABSTRACTION**

Abstraction is the methodology of hiding the implementation of internal details and showing the functionality to the users.

In Page Object Model design pattern, we write locators (such as id, name, Xpath etc.,) and the methods in a Page Class. We utilize these locators in tests, but we can’t see the implementation of the methods. Literally we hide the implementations of the locators from the tests.

Example: **Login function of LMS (Login Page Module) portal**

A page Class “Login Page” was created to store all the objects or locators of Login Page Module. All functions performed on the UI of Login page are stored as methods in the same Page.

**2. INTERFACE**

WebDriver is an Interface.

WebDriver driver = new ChromeDriver();

WebDriver driver = new FireFoxDriver();

Here, we are initializing Chrome browser using Selenium WebDriver. It means we are creating a reference variable (driver) of the interface (WebDriver) and creating an Object. Here WebDriver is an Interface as mentioned earlier and Chrome driver is a class.

**3. INHERITANCE**

The mechanism in Java by which one class acquires the properties (instance variables) and functionalities of another class is known as **Inheritance.**

We create a Base Class in the Automation Framework to initialize WebDriver interface, WebDriver waits, Property files, Excels, etc., in the Base Class. We extend the Base Class in other classes such Tests and Utility Class.

4. **POLYMORPHISM**

Polymorphism allows us to perform a task in multiple ways.

**METHOD OVERLOADING (Compile time Polymorphism)**

A class having multiple methods with same name, but different parameters is called Method Overloading

We use Implicit wait in Selenium. Implicit wait is an example of overloading. In Implicit wait we use different time stamps such as SECONDS, MINUTES, HOURS etc.

driver. manage (). timeouts (). implicitlyWait (30, TimeUnit.SECONDS).

driver. manage (). timeouts (). implicitlyWait (30, TimeUnit.MINUTES).

driver. manage (). timeouts (). implicitlyWait (30, TimeUnit.HOURS).

**Assert class** in TestNG is also an **example** of **overloading.**

**we have utility methods in our framework we used method overloading for setting up test data.**

**we can pass parameters set up with diferent parameter.**

**METHOD OVERRIDING (Run time polymorphism)**

We use a method which was already implemented in another class by changing its parameters. To understand this, you need to understand Overriding in Java.

webdriver we pass chrome driver or fireforx based on build we change the parameter, browsert type

Declaring a method in child class which is already present in the parent class is called Method Overriding. Examples are **get** and **navigate** methods of different drivers in Selenium.

**5. ENCAPSULATION**

All the classes in a framework are an example of Encapsulation.

In POM classes, we declare the data members using @FindBy and initialization of data members will be done using Constructor to utilize those in methods. Encapsulation is a mechanism of binding code and data (variables) together in a single unit.

settting the password we dont give getter methods

**Rest Assured** from Scratch: -

Add REST Assured and TestNG dependencies to the project.

**Given** ( ) 'Given' keyword, lets you set a background, here, you pass the request headers, query, and path param, body, cookies. This is optional if these items are not needed in the request.

**When** ( ) when' keyword marks the premise of your scenario. For example, 'when' you get/post/put something, do something else.

**Method** ( ) Substitute this with any of the CRUD operations(get/post/put/delete)

**Then** ( ) Your assert and matcher conditions go here.

**RequestSpecification** in Rest Assured can be used to group together common request specs and turn them into a single object. This is interface has methods for defining the base URL, base path, headers, and other parameters. To obtain a reference for **RequestSpecification**, we must use the given ( ) function of the Rest Assured class. We can't make an object out of RequestSpecification because it's an interface.

Its implemented class is RequestSpecificationImp.

Request Specification reqSpec = RestAssured.given ( ).

reqSpec.baseUri ("http: / /localhost: 8080")

reqSpec.basePath (" /employees");

**What are the best practices for developing a maintainable Rest Assured**

**Framework?**

1) Separate test data, test logic, and assertions using a modular approach.

2) Reduce code duplication by implementing reusable helper methods or classes.

3) Variable and method names should be meaningful and descriptive.

4) To make troubleshooting easier, implement effective error handling and logging.

|  |  |
| --- | --- |
| **List** | **Set** |
| An ordered collection of elements | An unordered collection of elements |
| Preserves the insertion order | Doesn’t preserve the insertion order |
| Duplicate values are allowed | Duplicate values are not allowed |
| Any number of null values can be stored | Only one null value can be stored |
| ListIterator can be used to traverse the List in any direction | ListIterator cannot be used to traverse a Set |
| Contains a legacy class called vector | Doesn’t contain any legacy class |

**How to install the Git plugin in Jenkins**

***Step 1***: Launch *Jenkins* and redirected to the ***Jenkins Dashboard***. Now click on the "***Manage Jenkins.*** "

***Step 2***: As soon as we will click on the ***Manage Jenkins*** link, we will be redirected to the *Manage Jenkins page*. Now click on the "***Manage Plugins.*** "

***Step 3***: As soon as we click on the "***Manage Plugins***" link, we will be redirected to the "***Plugin Manager***" page.

* ***If the git plugin is not already installed,****then click on the "****Available****" tab.*
* *Type "****Git Plugin****" in the Filter box and select the****Git Plugin****checkbox after its appearance.*
* *Choose either the "****Install without restart****" or "****Download now and install after restart****" button. If we will choose the "****Install without restart****" option then it will install the plugin without restarting Jenkins and if we choose "****Download now and install after restart****" then the Git plugin will be installed after restarting Jenkins.*

**Step 4**: Now after installing *Git Plugin*, we need to verify it also.

* *Click on the "****Installed****" tab.*
* *Type "****Git Plugin****" in the Filter box. (Once we verify it Appear Installed)*

**How to pull a GitHub repository in Jenkins?**

***Step 1***: Go to ***Jenkins Dashboard*** and click on the "***New Item*** " link to create a new job

***Step2***

* *Enter the Project Name.*
* *Select the Project type as we selected "Freestyle Project".*
* *Click on the OK button.*

**Step 3**: As soon as we click on the "***OK*** " button then we will be redirected towards the ***Configuration*** page. Here we can put the description of our project.

***Step 4:*** Now just scroll down and go to the ***Source Code Management*** section. Now, select the "***Git*** " option.

***Step 5***: Now enter the ***repository URL***

***6th Step***: Now, go to the ***Build triggers section*** and select the option "***GitHub hook trigger for GITScm polling***".

***Step 7***: Go to the ***Build*** section. Now click on "***Execute Windows batch Command*** " after clicking on the ***Add build*** step dropdown. We are using the ***Execute Windows Batch Command***

***Step 8***: Now here we will write batch commands to execute this *Java* Program.

cd <locate the package for source code>

javac Basic/Hello\_ToolsQA.java

java Basic/Hello\_ToolsQA

Now click on the ***Save*** button

***How do I trigger a build automatically in Jenkins?***

***Step 1*: Go to the *GitHub* repository and click on the *settings.***

***Step 2*: Click on the *Webhooks* option listed.**

***Step 3*: As soon as we click on *Webhooks*, we will redirect to the *Webhooks* page. Now, click on the "*Add webhook*” button (give information regarding webhooks.)**

***Step 4*: need to fill in is "*Payload URL.* need to give the *Jenkins URL*.**

***Step 5***: Now perform the below steps to setup webhooks in GitHub.

* *Put the Payload URL in the textbox. Kindly note that doesn't forget to append text****GitHub-webhook/****at the last.*
* *Click on the "****Just the push event****" option.*
* *Please make sure that you check the "****Active****" checkbox.*
* *Click on the "****Add webhook****” button.*
* After clicking on the ***Add webhook*** button, we will see a successful message.

***Step 6***: Now go to code, make some changes, and commit our code again. As soon as we commit out changes, webhook sent the notification to *Jenkins*, and automatically build will be triggered.

***Step 7***: After successfully running the build, we can see results in the console output.

***Step 8:*** After a successful run, we can see the blue sign under the *Build History* section highlighted inside the red rectangle in the below image. After that, click on "***Console Output*** " to see the actual output.

**To configure AWS credentials in Jenkins:**

* On the Jenkins dashboard, go to *Manage Jenkins > Manage Plugins* in the *Available* tab. Search for the Pipeline: AWS Steps plugin and choose Install without restart.
* Navigate to *Manage Jenkins > Manage Credentials > Jenkins (global) > Global Credentials > Add Credentials*.
* Select Kind as AWS credentials and use the ID & Credentials.
* Enter the access key ID and secret access key and choose OK.
* Create Amazon S3 buckets for each Region in the pipeline. S3 bucket names must be unique within a partition:
* Create a file named Jenkins file at the root of the project and add.
* Commit and push the code to the GitHub repository by running following commands:

git commit -am “Adding Jenkins pipeline config.”  
git push origin -u main.

Next, create a Jenkins Pipeline project:

1.From the Jenkins dashboard, choose New Item, select Pipeline, and enter the project name sam-jenkins-demo-pipeline.

2.Under Build Triggers, select *Poll SCM* and enter \* \* \* \* \*. This polls the repository for changes every minute

3.Under the Pipeline section, select Definition as Pipeline script from SCM.

* Select GIT under SCM and enter the repository URL.
* Set *Branches to build* to \*/main.
* Set the Script Path to Jenkins file

4.Save the project.

After the build finishes, we can see the pipeline.

How To Send Email Notification In Jenkins?

**Using Email Extension Plugin**

**Using Default Email Notifier**

**Email Extension Plugin**

*Step 1: Log in to the Jenkins Homepage*

*Step 2: Install Email Extension Plugin*

**Manage Jenkins-> Manage Plugins**

In the available tab search for Email Extension Plugin

*Step 3: Configure System*

Now go to **Manage Jenkins-> Configure System**.

*Step 4: Create Jenkins Pipeline Job*

Now go to Jenkins homepage and create a new job. Name the job with whatever name that you want and select pipeline. Click on OK.

*Step 5: View Console Output*

 Click on Build Number “#1” and click on “Console Output” on the build menu

*Step 6: Check Email.*

**Quality Assurance Vs Quality Control**

**QA** generally focuses on the processes and procedures that increase quality, such as training, documentation, monitoring, and audits. **QC** focuses on the product to identify faults that remain after development. QC specialists discover these vulnerabilities through a variety of methods, including software testing and beta or canary testing.

Spring boot test application framework with selenium testing cucumber

In our last project we built entire framework spring boot.

it saved lot of problems with dependencies etc.

We have WebDriver Beans **for chome,firefox and edge**. We inject the environment variable using @**value** annotation. this is where we use spring profiles for environment specific things.

Our webdriver bean cannot be singleton we need to use spring boot simplethread scope. As this will be an issue with parallel execution.

Since we need webdriver instance for every execution of the test.

and we use **ConditionalOnProperty** annotation with **name** and **havingValue** to set correct webdriver based on **name=browser** **and havingValue=Chrome**

we use the spring dependency for injecting the dependent page. Lets say we have a page which also needs two other pages. we create bean for those.

we use the page object model we define the elements **@FindBy** annotation and Pass how keyword and using key word. we use **PageFactory.initElements** method is used to initialize web elements.

we used the class **AbstractTestNGSpringContextTests** includes the spring **ApplicationContext**. To make it available when executing TestNG test, **AbstractTestNGSpringContextTests** has methods annotated with TestNG annotations like @BeforeClass and @BeforeMethod.

we annotated that with @**SpringBootTest.**

we have common logic for setting up the webdriver etc in that class we invoke the **springTestContextPrepareTestInstance.**

we have testng.xml and we can specify **parallel=true** and set **threadcount** if we need run the classes or methods parallely.

we use cucumber as our BDD framework.

we write feature file which has given when then keyworks with multiple scenarios and create respective **Step defintions** methods with **@Given @when and @And** annotation. This is where again dependency injection will be super easy where we can inject different pages.

for cucumber test runner we extend **AbStractTestNgCucumberTests** class and specify feature and glue parameters. Glue where we give the path to steps.

We create the **CucumberTestContextConfig** and annotate with @**CucumberContextConfiguration** and @**SpringBootTest**

We use hooks for our teardown logic at scenario level and this where we perform if scenario is failed take screenshot do our retry logic if we need to retry.

As I said before we can run through with command line we can specify our testrunner class.

We pass the data between steps using @**ScenarioScope**

we use maven surefile plugin to run in command line and thats the same command we configure in jenkins.

mvn clean test and pass the properties -Dbrowser=chrome

**Playwright:**

* I used Playwright to automate end-to-end testing in online banking applications. This includes scenarios such as logging in, transferring funds, **checking account balances**, and **verifying transaction histories.**
* **playwright’s** multi-browser support ensures consisent testing across different browser environments, ensuring a seamless user experience for customers using various browsers.
* In the E Commerce Playwright's device emulation capabilities enable testing the responsiveness of web applications on various **screen sizes** and **resolutions.**
* Playwright can be used to conduct performance testing by simulating many users engaging with the e-commerce platform simultaneously.

In this I will be doing **Performance Testing’s like Load Testing, Stress Testing, Scalability Testing, Response Time Testing.**

* The eCommerce project involves the development and maintenance of an online platform for buying and selling products or services.
* Validate the proper creation, modification, and cancellation of orders. Ensure that the order information is accurately recorded in the system and reflects **real-time updates**.
* Validate transactions, **payment processing**, **order management**, and **inventory** synchronization within the **POS** environment.
* **API** and **microservices** testing with **Karate** allows e-commerce companies to assess the scalability and performance of their systems under different load conditions.
* Verify the integration of the POS system with other components of the E-commerce **ecosystem**, such as the **online store**, payment gateways, and inventory management systems. Ensure seamless data flow and communication.
* Test the synchronization of inventory levels with **POS** transactions. Verify that product quantities are updated in real-time to prevent overselling or stockouts.
* Perform comprehensive testing across various browsers and devices to ensure a consistent and optimized user experience for **E-commerce** and **POS** interactions.
* The platform includes features such as product catalog, shopping cart, user accounts, checkout, payment processing, and order management.
* Design and implement automated test scripts using tools like **Selenium**, or similar frameworks.
* Collaborate with developers, product managers, and business analysts to understand requirements and translate them into effective test cases.
* Collaborate with security teams to conduct security testing on **E-commerce** and **POS** applications, identifying and addressing vulnerabilities in the system.
* Integrate automated tests into the Continuous Integration/Continuous Deployment (CI/CD) pipeline for quick validation of code changes.
* Conduct regression testing to identify and address potential issues in new releases.
* Perform cross-browser and cross-device testing to ensure a consistent user experience.
* Validate user authentication, authorization, and role-based access control.
* Execute performance testing to ensure the platform can handle expected and peak loads.

**1. Continuous Integration (CI):** - Developers work on their code locally, making changes and improvements.  
 - As soon as they're ready to integrate their changes into the main codebase, they push their code to a version control system (like Git).  
 - Upon pushing the changes, the CI server (such as Jenkins, Travis CI, CircleCI, etc.) detects the new code and triggers an automated build process.  
 - During the build process, the CI server pulls the latest code from the repository, compiles the code, runs automated tests, and performs other necessary checks.  
 - If any issues are detected during this process (such as failing tests or compilation errors), the CI server notifies the developers immediately, allowing them to address the problems promptly.  
 **2. Continuous Deployment (CD):** - Once the CI process is successfully completed and the code passes all tests, it's ready for deployment.  
 - In a CD pipeline, the CI server can automatically deploy the code to various environments (like development, staging, or production) based on predefined rules and configurations.  
 - Deployment can involve various steps such as packaging the application, provisioning infrastructure (if necessary), configuring services, and deploying the application.  
 - Automated deployment ensures consistency and reliability, reducing the risk of errors that may occur during manual deployment processes.  
 - Continuous Deployment pipelines often include additional steps like smoke testing or integration testing in the deployed environment to ensure that the application behaves as expected in its production environment.