**Assessment – SS&C**
Marija (Masha) Smolnikova
Date: June 3, 2025
**Project Repository:** github.com/bessmash/ssc-submission

**Question 1: Test Strategy & Planning**
**Describe how you would approach creating a test strategy for a new enterprise-level web application. What key components would you include, and how would you ensure alignment with Agile development practices?**

When I hear “a new web application,” I imagine a product that includes AI in some form. So for this answer, I’m considering an AI-driven enterprise-level web application. My overall approach would combine risk-based prioritization, early QA involvement, and CI/CD integration from the start.

First, I would work with the team to clarify test objectives. Are we validating functional coverage, AI model behavior, user experience, or regulatory compliance? Most likely, all of these. For enterprise-level applications, objectives usually also include security, scalability, and integration, especially with third-party services like authentication providers, document systems, or internal support tools. These goals shape the foundation for defining the overall scope of testing.

Next, I’d identify the testing scope: static testing of requirements and compliance, system-level integration testing, exploratory testing, regression, and performance testing as the product evolves. For AI components, I’d include validation of model behavior using monitored datasets, checking for bias, and ensuring reproducibility through logging and versioning. These areas require clear definitions over time but must still follow structured documentation and traceability like any other part of the system.

For test design, I tend to blend classical techniques, like equivalence partitioning and boundary analysis, with exploratory sessions during unstable builds, and sometimes paired testing with developers. I would adjust automation scope and timing based on sprint structure, identified risks, and test coverage gaps.

Tool-wise, I would start early conversations with DevOps to ensure staging and production are closely mirrored. I would flag dependencies like test data provisioning (whether we’d need dynamic data generation, masked production copies, or pre-seeded databases), as well as access control and integration with CI/CD or test management systems. This would help avoid late surprises and keep environments predictable for testing.

To stay Agile, I’d document the strategy in a format like Confluence so it’s easy to share and update. I would use refinement meetings to check what’s testable and whether priorities have shifted. If new stories were added, new features introduced, or the team’s focus changed, I would update the strategy document accordingly to reflect those changes and keep everyone in the loop.

**Question 2: Automation Framework Design**
**Explain the difference between a data-driven and a keyword-driven automation framework. Which would you choose for testing a financial web application and why?**

A data-driven test approach focuses on separating test logic from test data. The same test script is run multiple times using different sets of input values, often pulled from external sources like CSV files, Excel sheets, or structured objects. This approach is efficient when the steps of the automation test remain constant, but the input variations are numerous. It usually works well for form validation, edge case coverage, negative test scenarios, and large-scale data input. The main purpose of a data-driven framework is to make it easy to test the same logic with different sets of data, allowing testers to change or switch input files without needing to modify the original test scripts.

A keyword-driven framework is structured around modular, reusable functions or methods, each represented by a keyword that describes a specific action or behavior. These keywords are typically organized into a shared library, and test cases are written by combining them into readable sequences. This structure supports clearer test design and helps make test scenarios understandable to non-technical stakeholders, allowing them to participate in defining test logic without needing to write code. Tools like Robot Framework, Cucumber, and Concordion follow this model, making it easier to express tests in business-readable language while executing automation scripts behind the scenes.

To summarize, a data-driven framework is best when a QA engineer wants to test the same functionality with various sets of data, while a keyword-driven framework focuses on enabling non-technical stakeholders to create and run tests without having to learn how to code.

For testing a financial web application, I would choose to build a **hybrid framework that combines both**: the readability of a keyword-driven approach and the scalability of a data-driven one. Financial applications often require testing a wide variety of input combinations across multiple user roles and access levels, while also supporting collaboration with teams outside QA. I would introduce a data-driven layer into the keyword-driven framework on a scale larger than the standard table-based approach typically available by design. This setup would support efficient, high-coverage testing across realistic scenarios, while also improving traceability and making the audit process easy for non-technical stakeholders.

This setup would also support long-term maintenance of the application. As the application and business logic evolve, test steps would remain easy to update, while data variations could expand without rewriting test logic. For a regulated, multi-user, high-risk environment like a financial platform, that kind of balance would be valuable.

**Question 3: API Testing**
**You are tasked with testing a RESTful API for a banking application. What steps would you take using Postman and Rest Assured to validate both functional and security aspects of the API?**

Postman and the RestAssured automation framework together provide thorough coverage of both the functional behavior and security of the banking application's API. I would use Postman primarily for manual and exploratory testing, while Java + RestAssured, executed with Serenity Runner, in my opinion, supports scalable, CI-integrated automation. This combination of tools also makes it easier to add reporting and track test metrics later on.

While functional and security API tests both operate on the same endpoints, they serve different purposes. **Functional testing** ensures that the API behaves correctly for different inputs and returns the appropriate status codes.**Security testing** focuses on protecting access, enforcing authorization, blocking injection attempts, and handling cases like missing session tokens or too permissive CORS headers.

To demonstrate both types of validation, I selected three representative tests from each category and implemented them using both **Postman** and Java with **RestAssured**. For demonstration purposes, I used the publicly available banking web application at ParaBank as the system under test.

**Functional Tests:**

- Login Test
  ***Postman*****:** Sends POST /login.htm, expects 200 OK, verifies presence of "Accounts Overview" in response.
  ***RestAssured*****:** Logs in, confirms redirect (302), captures session, follows-up to the dashboard and verifies 200 OK and presence of "Account Overview" text in response.

- Get User Data Test
  ***Postman*****:** Calls GET /login/{username}/{password} and verifies 200 OK and firstName == John; userId is extracted and reused.
  ***RestAssured*****:** Verifies 200 OK and the same firstName. User ID is returned and matches expected structure and name formatting.

- Get Account Balance Test
  ***Postman*****:** Sends request to GET /customers/{userId}/accounts, expects 200 OK, and checks for "CHECKING" account.
  ***RestAssured*****:** Verifies 200 OK and the same firstName. User ID is returned and matches expected structure and name formatting.

**Security Tests:**

- Login over HTTP
  ***Postman*****:** Attempts login using http:// instead of https://, expects failure and response indicating misconfiguration or rejection.
  ***RestAssured*****:** Verifies that status code is not 200 or 302

- SQL Injection in User ID
  ***Postman*****:** Sends userId + ' OR '1'='1 in the path; expects non-200 status and no sensitive data leakage.
  ***RestAssured*****:** Sends the same malicious ID and verifies that the system responds with 403 Forbidden.

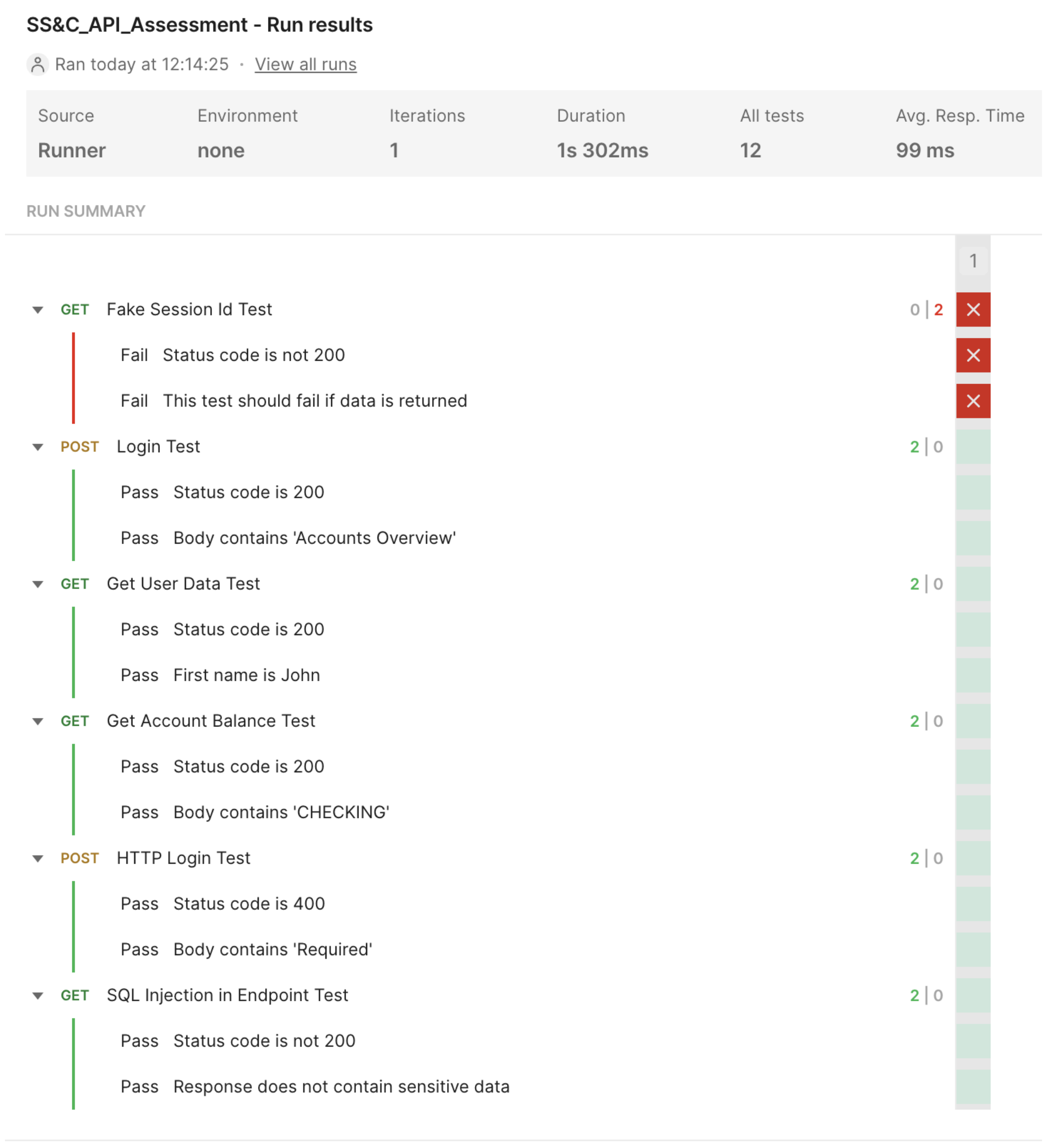
- Sessionless Access Attempt **(FAIL)**
  ***Postman*****:** Uses a fake JSESSIONID and attempts to access account data. In the current environment, the test fails (data returned).
  ***RestAssured*****:** Sends the valid request with no session; the test fails as 200 is returned instead of 403.
  **Security Finding: This confirms a real vulnerability where user data can be accessed without a session control.**

The two screenshots included at the end of this answer demonstrate full test execution in both environments. The Postman run shows validation of all six test cases, and the JUnit output in Eclipse confirms successful execution of the same tests using RestAssured under Serenity Runner. Together, these tools allowed me to validate both functional behavior and critical security controls in a way that is transparent, repeatable, and ready to evolve into a more structured automation solution.

The two screenshots at the end of this answer show full test execution in both environments. The Postman run demonstrates all six test cases, including the expected failure of the sessionless access attempt. The JUnit output in Eclipse shows the same results under Serenity Runner. Together, these tools made it possible to validate both functional behavior and key security checks.

**Screenshot:** Postman test run

**Collection JSON file:** SS&C_API_Assessment.postman_collection.json



**SS&C_API_Assessment - Run results**

Ran today at 12:14:25 · View all runs

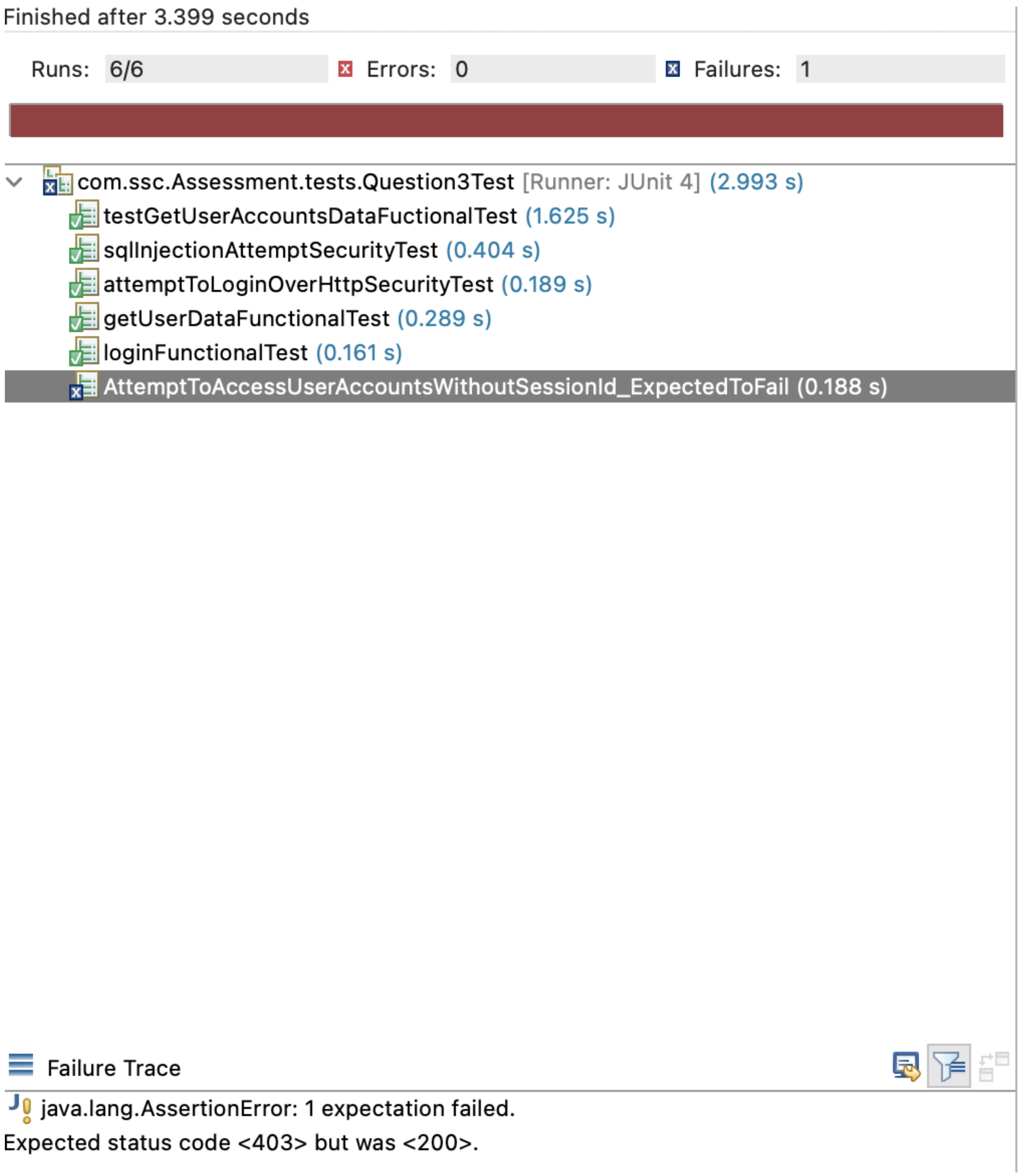
| Source | Environment | Iterations | Duration | All tests | Avg. Resp. Time |
|---|---|---|---|---|---|
| Runner | none | 1 | 1s 302ms | 12 | 99 ms |

RUN SUMMARY

| | | 1 |
|---|---|---|
| GET Fake Session Id Test | 0 \| 2 | ✗ |
| Fail Status code is not 200 | | ✗ |
| Fail This test should fail if data is returned | | ✗ |
| POST Login Test | 2 \| 0 | 🟩 |
| Pass Status code is 200 | | 🟩 |
| Pass Body contains 'Accounts Overview' | | 🟩 |
| GET Get User Data Test | 2 \| 0 | 🟩 |
| Pass Status code is 200 | | 🟩 |
| Pass First name is John | | 🟩 |
| GET Get Account Balance Test | 2 \| 0 | 🟩 |
| Pass Status code is 200 | | 🟩 |
| Pass Body contains 'CHECKING' | | 🟩 |
| POST HTTP Login Test | 2 \| 0 | 🟩 |
| Pass Status code is 400 | | 🟩 |
| Pass Body contains 'Required' | | 🟩 |
| GET SQL Injection in Endpoint Test | 2 \| 0 | 🟩 |
| Pass Status code is not 200 | | 🟩 |
| Pass Response does not contain sensitive data | | 🟩 |

**Screenshot:** JUnit test run in Eclipse

**Test code:** Question3Test.java



```
Finished after 3.399 seconds

Runs: 6/6    Errors: 0    Failures: 1

com.ssc.Assessment.tests.Question3Test [Runner: JUnit 4] (2.993 s)
    testGetUserAccountsDataFuctionalTest (1.625 s)
    sqlInjectionAttemptSecurityTest (0.404 s)
    attemptToLoginOverHttpSecurityTest (0.189 s)
    getUserDataFunctionalTest (0.289 s)
    loginFunctionalTest (0.161 s)
    AttemptToAccessUserAccountsWithoutSessionId_ExpectedToFail (0.188 s)

Failure Trace
java.lang.AssertionError: 1 expectation failed.
Expected status code <403> but was <200>.
```

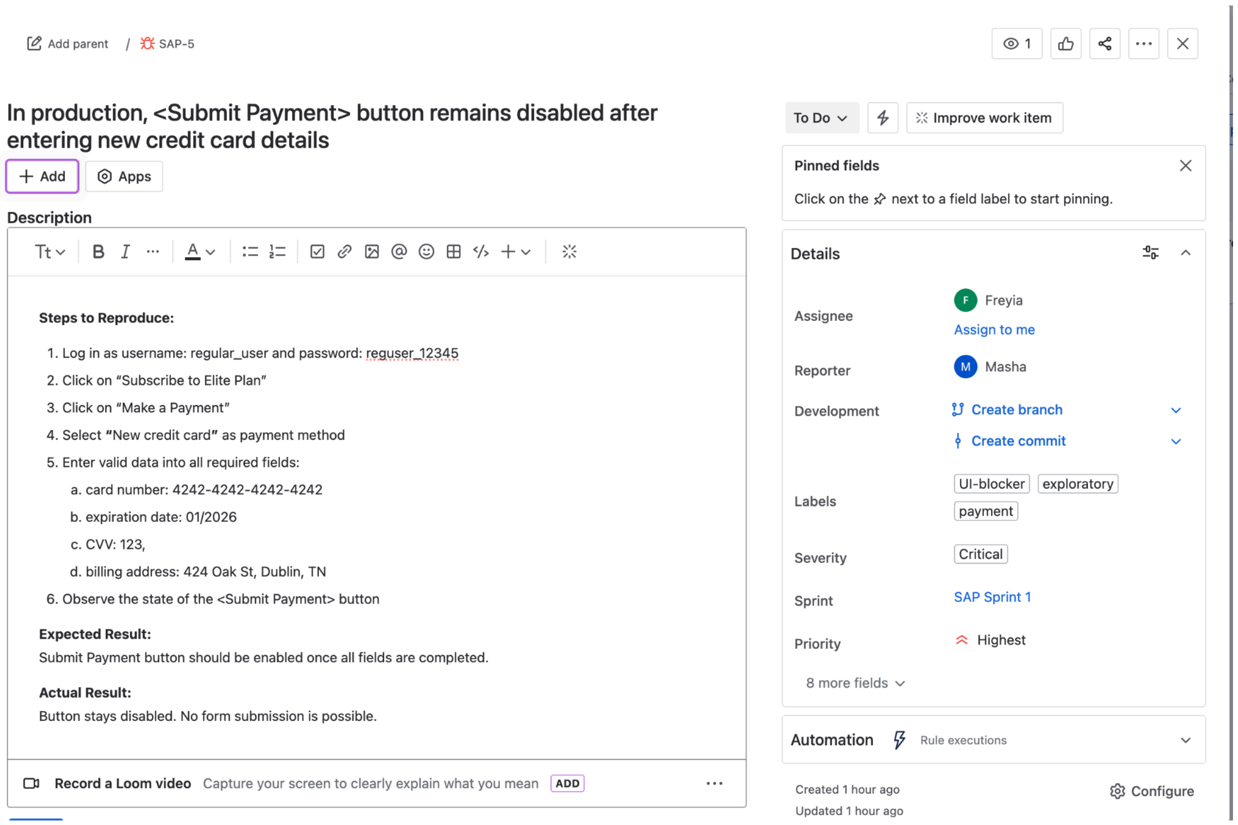
## Question 4: Defect Management

**A critical defect is found in production. Walk through how you would use Jira/Xray to manage the defect lifecycle, from discovery to resolution, including communication with stakeholders.**

To demonstrate this, I set up a real Jira/Xray environment and walked through a full defect lifecycle with the following roles: QA (***Masha***), Developer (***Freyia***), and Product Owner (***Andy***). The process shows how I manage urgent production issues in an Agile setting while maintaining traceability, structure, and communication.

1. The sprint began with an exploratory testing task: **Ad Hoc Test Execution**, assigned by ***Andy*** based on support team feedback. ***Masha*** completed it, and during that session, she found a production-critical issue**(!)**: When a user fills out all new credit card fields, the <Submit Payment> button remains disabled.
2. She created a **Bug**, documented all reproduction steps, set severity to *Critical*, assigned it to ***Freyia***, and notified ***Andy***. ***Andy*** reviewed and added the bug to Sprint 1.

**Screenshot:** Bug Logged in Jira



In production, <Submit Payment> button remains disabled after entering new credit card details

Description

**Steps to Reproduce:**

1. Log in as username: regular_user and password: reguser_12345
2. Click on "Subscribe to Elite Plan"
3. Click on "Make a Payment"
4. Select **"**New credit card**"** as payment method
5. Enter valid data into all required fields:
   a. card number: 4242-4242-4242-4242
   b. expiration date: 01/2026
   c. CVV: 123,
   d. billing address: 424 Oak St, Dublin, TN
6. Observe the state of the <Submit Payment> button

**Expected Result:**
Submit Payment button should be enabled once all fields are completed.

**Actual Result:**
Button stays disabled. No form submission is possible.

Record a Loom video Capture your screen to clearly explain what you mean ADD

Pinned fields
Click on the next to a field label to start pinning.

Details

| | |
|---|---|
| Assignee | Freyia<br>Assign to me |
| Reporter | Masha |
| Development | Create branch<br>Create commit |
| Labels | UI-blocker, exploratory, payment |
| Severity | Critical |
| Sprint | SAP Sprint 1 |
| Priority | Highest |

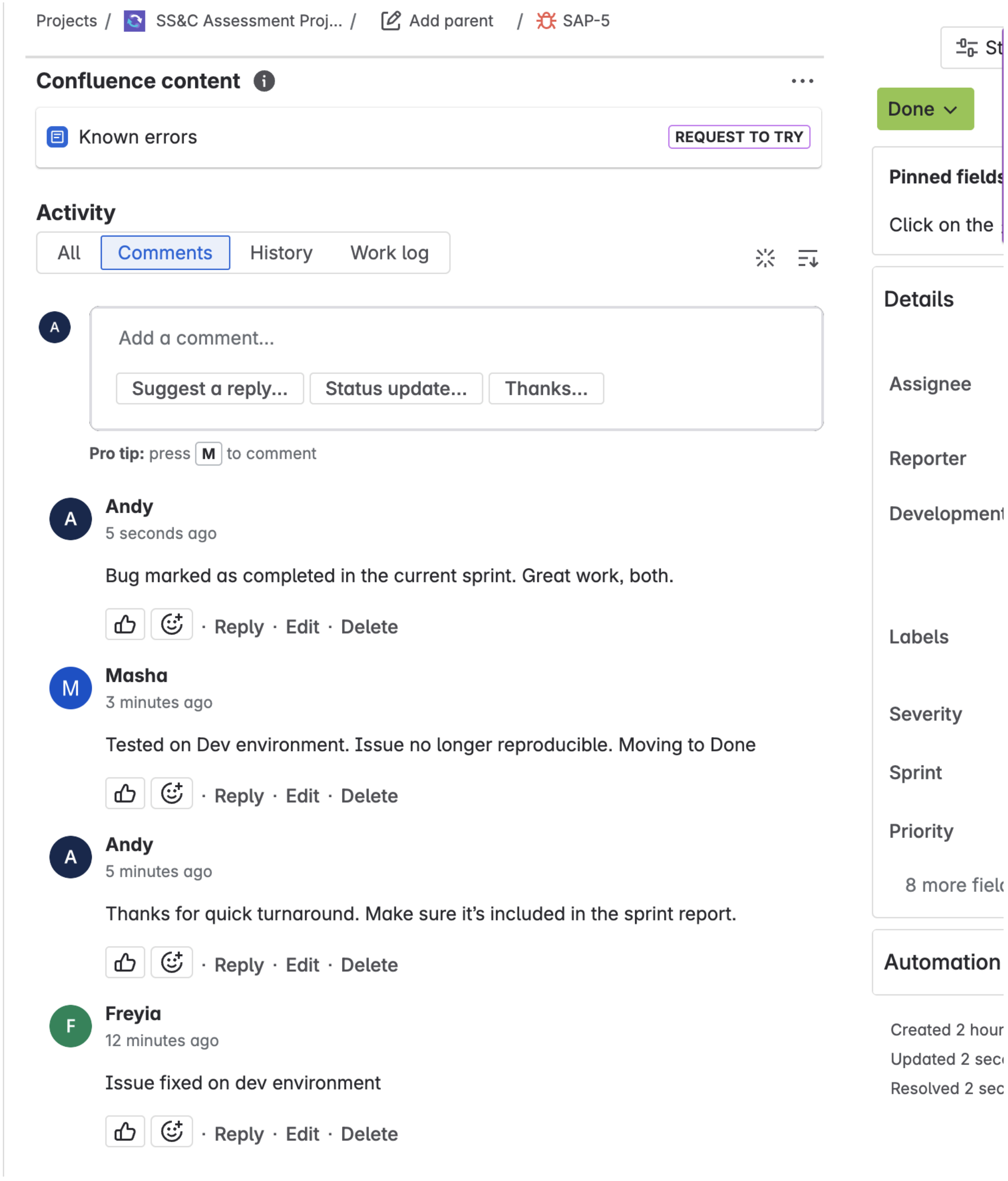
8 more fields

Automation Rule executions

Created 1 hour ago
Updated 1 hour ago
Configure

3. ***Freyia*** fixed the issue in the Dev environment, added a comment, and reassigned it to ***Masha***. ***Masha*** verified the fix, confirmed it in the comments, and moved the issue to Done. ***Andy*** responded to close the loop.

**Screenshot**: [Communication between QA, Dev, and PO]



**Andy** 5 seconds ago
Bug marked as completed in the current sprint. Great work, both.

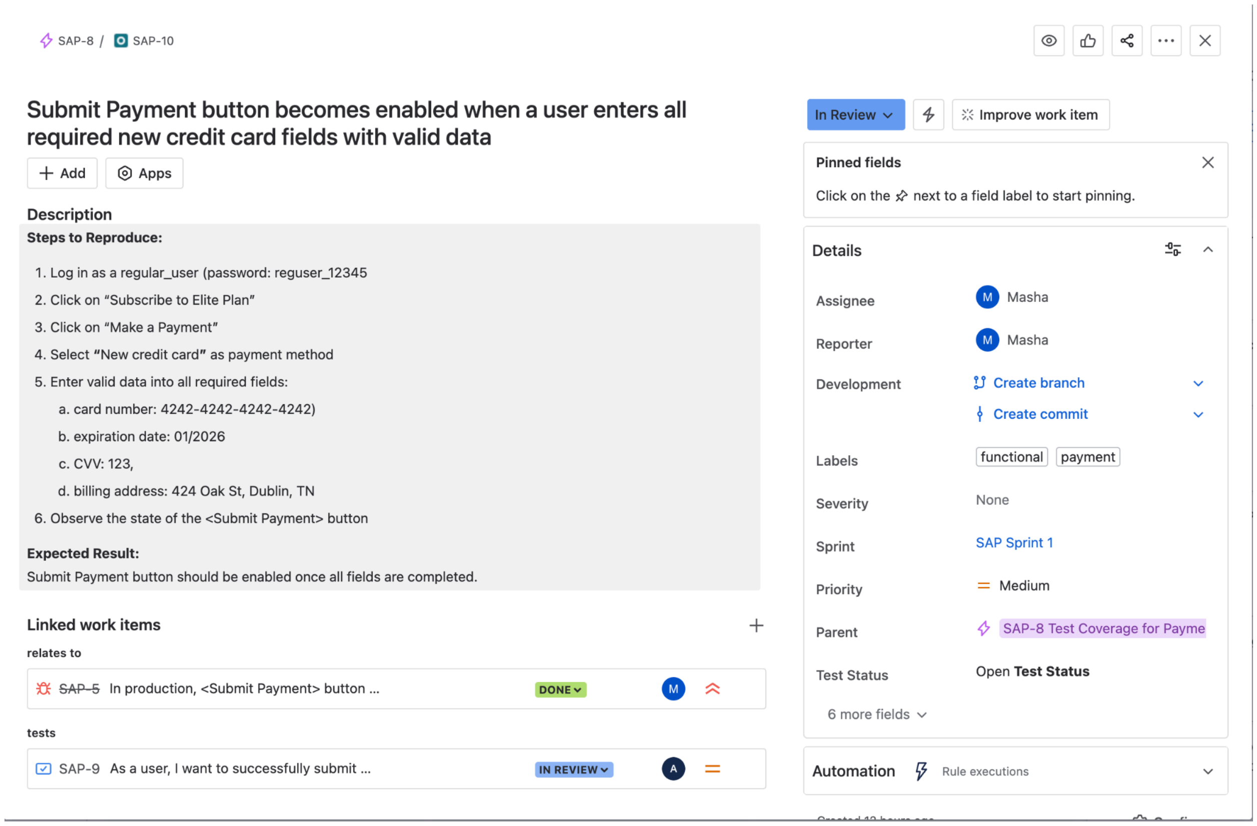
**Masha** 3 minutes ago
Tested on Dev environment. Issue no longer reproducible. Moving to Done

**Andy** 5 minutes ago
Thanks for quick turnaround. Make sure it's included in the sprint report.

**Freyia** 12 minutes ago
Issue fixed on dev environment

4. After resolution, ***Masha*** created a manual **Test Case** (SAP-10) in Xray based on the bug scenario. Due to a rendering issue (*sorry, couldn't fix it in time*), the test steps were added to the description instead of the designated steps section. The test case was linked to the bug, the related user story (SAP-9), and the Epic for payment test coverage (SAP-8).

**Screenshot:** [Xray Manual Test Case SAP-10]



Submit Payment button becomes enabled when a user enters all required new credit card fields with valid data

Description

**Steps to Reproduce:**

1. Log in as a regular_user (password: reguser_12345
2. Click on "Subscribe to Elite Plan"
3. Click on "Make a Payment"
4. Select "New credit card" as payment method
5. Enter valid data into all required fields:
   a. card number: 4242-4242-4242-4242)
   b. expiration date: 01/2026
   c. CVV: 123,
   d. billing address: 424 Oak St, Dublin, TN
6. Observe the state of the <Submit Payment> button

**Expected Result:**
Submit Payment button should be enabled once all fields are completed.

Linked work items

relates to

~~SAP-5~~ In production, <Submit Payment> button ... DONE M

tests

SAP-9 As a user, I want to successfully submit ... IN REVIEW A

Pinned fields

Click on the next to a field label to start pinning.

Details

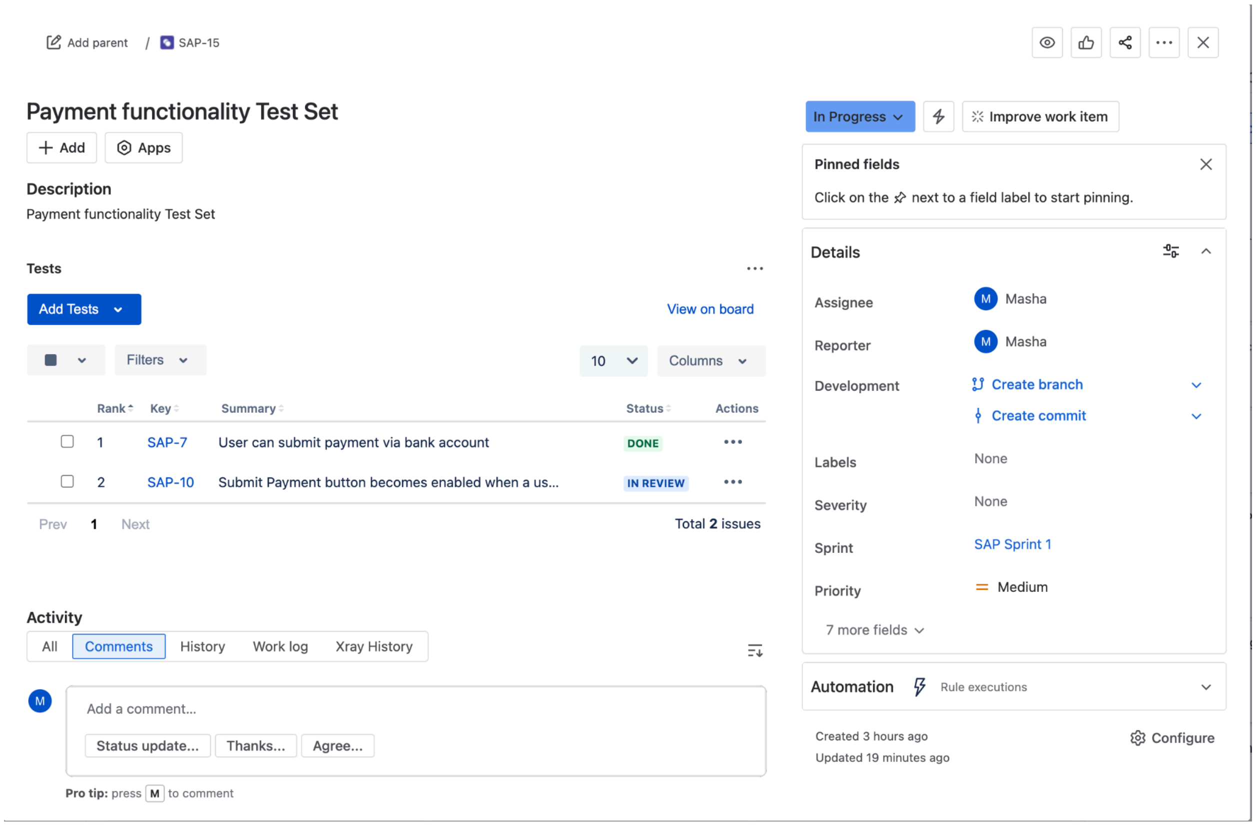
| | |
|---|---|
| Assignee | Masha |
| Reporter | Masha |
| Development | Create branch<br>Create commit |
| Labels | functional payment |
| Severity | None |
| Sprint | SAP Sprint 1 |
| Priority | Medium |
| Parent | SAP-8 Test Coverage for Payme |
| Test Status | Open **Test Status** |

6 more fields

Automation Rule executions

5. That test was added to a **Payment Functionality Test Set** (SAP-15) along with another case for bank payment.

**Screenshot:** [Payment Functionality Test Set]



Add parent / SAP-15

Payment functionality Test Set

Description

Payment functionality Test Set

Tests

| Rank | Key | Summary | Status | Actions |
|---|---|---|---|---|
| 1 | SAP-7 | User can submit payment via bank account | DONE | ••• |
| 2 | SAP-10 | Submit Payment button becomes enabled when a us... | IN REVIEW | ••• |

Prev 1 Next Total 2 issues

Activity

Add a comment...

Pro tip: press M to comment

In Progress

Pinned fields

Click on the next to a field label to start pinning.

Details

| | |
|---|---|
| Assignee | Masha |
| Reporter | Masha |
| Development | Create branch<br>Create commit |
| Labels | None |
| Severity | None |
| Sprint | SAP Sprint 1 |
| Priority | Medium |

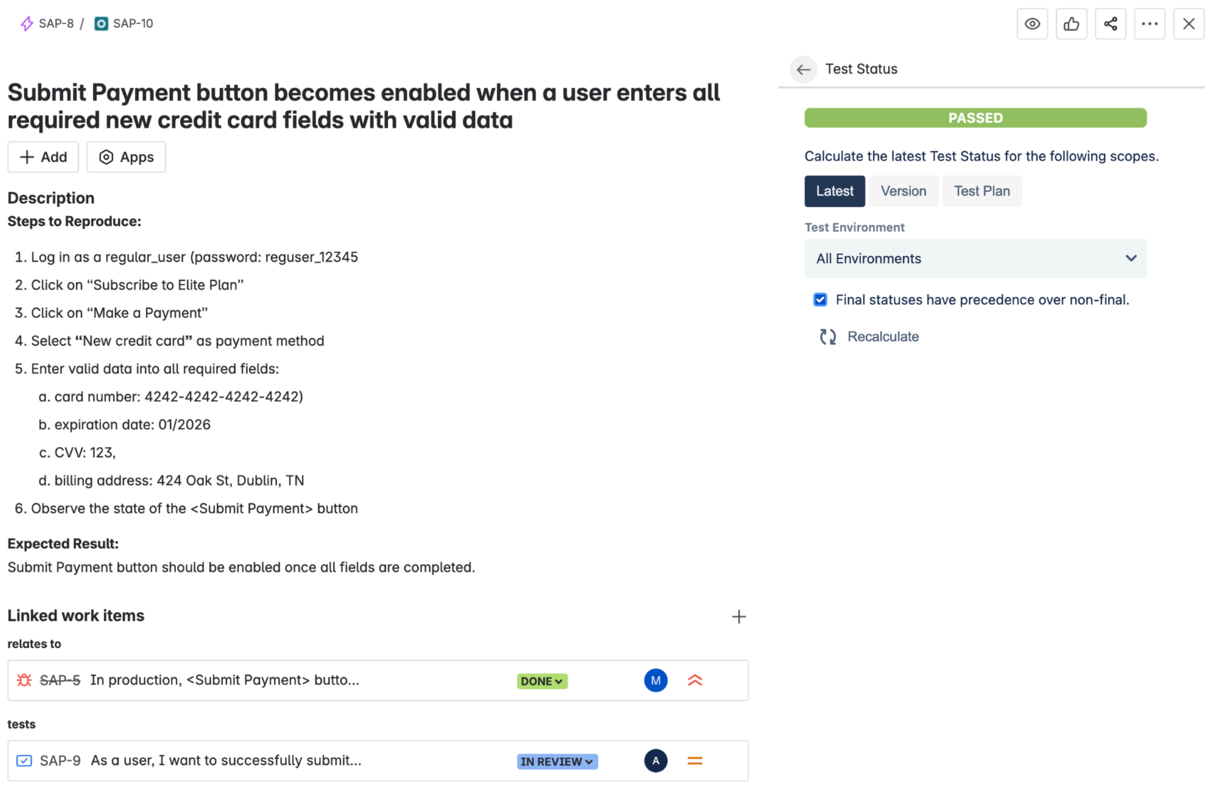
7 more fields

Automation Rule executions

Created 3 hours ago
Updated 19 minutes ago

6. Both the **Payment Functionality Test Set** and the original **Ad Hoc Execution** were included in a **Sprint 1 - QA Test Plan** (SAP-16), reflecting a shift-left QA approach and ensuring regression readiness. (*The Test Plan screenshot not shown due to the same Xray rendering issue*)
7. The test case was marked **Passed**, but intentionally left **In Review**, the team needed more time to observe behavior and hold the fix close.

**Screenshot:** Test Case marked Passed


SAP-8 / SAP-10

**Submit Payment button becomes enabled when a user enters all required new credit card fields with valid data**

**Description**

**Steps to Reproduce:**

1. Log in as a regular_user (password: reguser_12345
2. Click on "Subscribe to Elite Plan"
3. Click on "Make a Payment"
4. Select **"New credit card"** as payment method
5. Enter valid data into all required fields:
   a. card number: 4242-4242-4242-4242)
   b. expiration date: 01/2026
   c. CVV: 123,
   d. billing address: 424 Oak St, Dublin, TN
6. Observe the state of the <Submit Payment> button

**Expected Result:**
Submit Payment button should be enabled once all fields are completed.

**Linked work items**

relates to

~~SAP-5~~ In production, <Submit Payment> butto... DONE M

tests

SAP-9 As a user, I want to successfully submit... IN REVIEW A

Test Status

PASSED

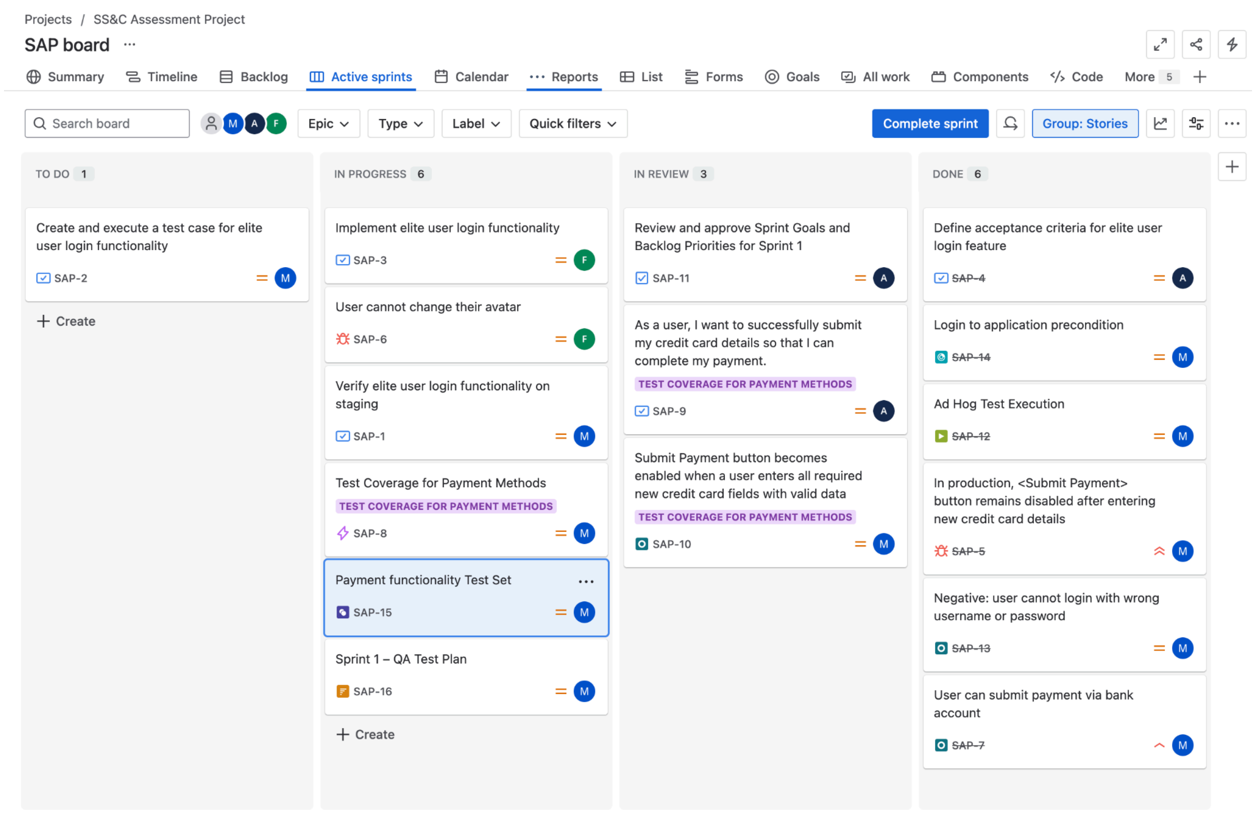
Calculate the latest Test Status for the following scopes.

Test Environment: All Environments

✓ Final statuses have precedence over non-final.

8. The **Sprint Board** shows the entire lifecycle in motion.

**Screenshot: Sprint Board**



Projects / SS&C Assessment Project

SAP board

TO DO 1

- Create and execute a test case for elite user login functionality — SAP-2 — M

IN PROGRESS 6

- Implement elite user login functionality — SAP-3 — F
- User cannot change their avatar — SAP-6 — F
- Verify elite user login functionality on staging — SAP-1 — M
- Test Coverage for Payment Methods — TEST COVERAGE FOR PAYMENT METHODS — SAP-8 — M
- Payment functionality Test Set — SAP-15 — M
- Sprint 1 – QA Test Plan — SAP-16 — M

IN REVIEW 3

- Review and approve Sprint Goals and Backlog Priorities for Sprint 1 — SAP-11 — A
- As a user, I want to successfully submit my credit card details so that I can complete my payment. — TEST COVERAGE FOR PAYMENT METHODS — SAP-9 — A
- Submit Payment button becomes enabled when a user enters all required new credit card fields with valid data — TEST COVERAGE FOR PAYMENT METHODS — SAP-10 — M

DONE 6

- Define acceptance criteria for elite user login feature — ~~SAP-4~~ — A
- Login to application precondition — ~~SAP-14~~ — M
- Ad Hog Test Execution — ~~SAP-12~~ — M
- In production, <Submit Payment> button remains disabled after entering new credit card details — ~~SAP-5~~ — M
- Negative: user cannot login with wrong username or password — ~~SAP-13~~ — M
- User can submit payment via bank account — ~~SAP-7~~ — M

**Question 5: Agile QA Practices**
**How do you ensure quality in a fast-paced Agile environment? Describe how you participate in Agile ceremonies and contribute to requirement refinement and test estimation.**

Usually, I focus on integrating quality early through a shift-left approach. I prefer to be involved during the conceptual phase, sometimes even before formal user stories are defined, so I can begin identifying potential test conditions from the initial test basis. This helps uncover ambiguities early and supports traceability from requirements to test coverage.

I don’t attend every meeting, but I prioritize the ones where real decisions are made, like architecture discussions, tool selection, or backlog refinement. In those, I think both as a user and as a QA. I ask myself what might go wrong, what test types will be needed, such as functional, non-functional, or exploratory, and what dependencies could affect test coverage.

During sprint planning, I contribute estimates and review the tasks from the QA perspective to make sure they are achievable. If I’m leading a team, I try to balance the workload so we meet expectations without adding unnecessary pressure. If time allows, we often do more than what was originally planned.

In daily stand-ups, I summarize the QA team's updates (defects logged, scripts created, test cases written, blockers encountered, etc.) to keep communication more focused and clear. I avoid bringing up interpersonal issues during group calls but address them quietly afterward. I use stand-ups to clarify risks and support rapid feedback loops. I also like to highlight team members' contributions by name when they did great.

By the time we enter sprint review and retrospectives, I come prepared with clear traceability between test cases and user stories, along with defect statistics, test coverage, and other relevant metrics. I also share my opinion on how well the team met the defined entry and exit criteria for testing activities during the sprint and suggest adjustments for the next one. If production defects occurred, I help map them back to missed test conditions or gaps in the requirements.

Throughout the sprint, I balance exploratory testing during unstable phases with structured manual testing and growing automation coverage as the application becomes more stable. I usually begin setting up the framework early, keeping it modular and layered, and connecting it to CI tools like Jenkins or GitHub Actions, even before full test scripts are in place. This way, by the time the first stable build is ready, the automation infrastructure for continuous feedback is already working.

**Question 6: Selenium + JavaScript Automation**
**Write a Selenium test script (in JavaScript or Java) that logs into a sample banking web app, navigates to the transactions page, and verifies that the latest transaction is displayed correctly.**

The script below is written in JavaScript using the official Selenium WebDriver bindings for Node.js. It automates login to the ParaBank demo banking app, navigates to the user's Accounts Overview, selects an account, and verifies that the latest transaction is displayed correctly. The test checks for a valid date, ensures one currency amount is shown, and confirms the description field is not empty.

To handle UI inconsistencies in the dashboard and account detail pages, the test retries the overview link when account links are incomplete and resiliently handles accounts that fail to load transactions by skipping them and continuing through the list. Browser behavior is controlled via environment variables. `TEST_HEADLESS=false` allows the test to run with a visible browser window, while `TEST_USERNAME` and `TEST_PASSWORD` let the user override login credentials if needed.

A Java version of the same test was also created using Selenium with Serenity Runner. It mirrors the logic and validations of the JavaScript implementation.

**JavaScript Implementation:** question6test.js

```
const { Builder, By, until } = require('selenium-webdriver');
const assert = require('assert');
async function runTest() {
       const base_url = "https://parabank.parasoft.com/parabank/index.htm";

       // should be parameterized from command line
       const username = process.env.TEST_USERNAME || 'john';
       const password = process.env.TEST_PASSWORD || 'demo';

       // parameter to run tests headless
       const headless = process.env.TEST_HEADLESS !== 'false'; // default is true

       // number of attempts to refresh accounts table
       let attemptCount = 3;
       // browser options
       const chrome = require('selenium-webdriver/chrome');
       let options = new chrome.Options();
       options.addArguments("--window-size=1920,1080");
       if (headless)
              options.addArguments("--headless");
       // driver
       const driver = await new Builder()
       .forBrowser('chrome')
       .setChromeOptions(options)
```

```
        .build();
        try {

        // go to the site
        await driver.get(base_url);
        const title = await driver.getTitle();
        assert(title.includes("ParaBank"), "Page title is incorrect");
        // login
        await driver.findElement(By.name('username')).sendKeys(username);
        await driver.findElement(By.name('password')).sendKeys(password);
        await driver.findElement(By.css('input.button')).click();
        // wait for account overview table
        await driver.wait(until.elementLocated(By.id('accountTable')), 10000);
               const overview = await driver.findElement(By.id('showOverview')).getText();
               assert(overview.includes("Accounts Overview"),
                      "Expected 'Accounts Overview' to appear, but got: " + overview);

        // getting accounts links
        let accountLinks = await driver.findElements(By.xpath("//td/a"));
        // click "Accounts Overview" again if there's only one account
        let attempts = 0;
        while (accountLinks.length <= 1 && attempts < attemptCount) {
               console.log("Refreshing the account list...");
               await driver.findElement(By.linkText('Accounts Overview')).click();
               await driver.sleep(5000);
               accountLinks = await driver.findElements(By.xpath("//td/a"));
               attempts++;
        }
        // Loop through accounts
        for (let i = 0; i < accountLinks.length; i++) {
        try {
               const accountId = await accountLinks[i].getText();
               await accountLinks[i].click();
                      // waiting for account id on a page
               await driver.sleep(5000);
               let accountDetails = await driver.wait(
                      until.elementLocated(By.id('accountId')), 5000);
               const textDetails = await
driver.findElement(By.id('accountDetails')).getText();
                      assert(textDetails.includes(accountId),
                             "Expected " + accountId + " to appear, but got: " +
textDetails);

               // getting the last transaction element from the table
               const transactionRow = await driver.findElement(
                      By.xpath("//table[@id='transactionTable']/tbody/tr[last()]")
               );
               const lastTransaction = await transactionRow.getText();
               console.log('Last transaction found:', lastTransaction);
```

```
              // verify row starts with a valid date
              assert(/^\d{2}-\d{2}-\d{4}.*/.test(lastTransaction),
                     "Row does not start with a valid date");
              // verify there is only one currency value in the string
              const countValues = (lastTransaction.match(/\$\d+\.\d{2}/g) || []).length;
                     assert.strictEqual(countValues, 1, "Unexpected number of currency
values");

              // check the description is not empty
              const descriptionStart = lastTransaction.indexOf(" ")+ 1;
              const amountStart = lastTransaction.lastIndexOf('$');
              const description = lastTransaction.substring(descriptionStart,
amountStart).trim();
                     assert(description.length > 0, "Description is empty");
              // exit from loop
              break;
       } catch (error) {
              console.log('Skipping glitchy account');
              await driver.navigate().back();
              await driver.wait(until.elementLocated(By.xpath("//td/a")), 5000);
              accountLinks = await driver.findElements(By.xpath("//td/a")); // re-fetch
       }
    }
  } catch (error) {
       console.log("No transactions found. "
                                   + "This may be expected for a new or inactive
account.");
  } finally {
    await driver.quit();
  }
}
runTest();
```

**Link to a Java version of the same test:** Question6Test.java

**Question 7: Gherkin/Cucumber Scenario**
**Write a Gherkin feature file for the following scenario:A user logs into an online banking portal, transfers money to another account, and receives a confirmation message.**

Below is a Cucumber feature file that describes this scenario in a business-readable format:

```
@question7
Feature: Money Transfer
 As a registered user of the Online Banking Portal
 I want to be able to transfer money to another account
 So that I can complete my personal transactions successfully
 Scenario: Successful money transfer to another account
   Given the user is logged into the Online Banking Portal
   When they navigate to the "Money Transfer" page
   And they enter "Account Number" and "Transfer Amount"
   And they submit the transfer request
   Then a confirmation message should be displayed
```

This feature is implemented in the ***`AssessmentProject`* Maven project** using Serenity and Cucumber. The Gherkin file is located in the standard resources directory and is linked below:

**Feature File:** question7.feature

**Step Definitions**: Question7Steps.java

**Test Runner**: Question7Test.java

**Question 8: WinAppDriver Test**
**Create a basic test case using WinAppDriver to automate a Windows-based calculator app. The test should perform an addition operation and verify the result.**

To automate a Windows-based calculator application, I used WinAppDriver and Java with JUnit. The test performs a simple addition operation (25 + 75) and verifies that the result displayed is 100. The calculator application is started via WinAppDriver using its application ID. Button clicks are simulated through findElementByName, and the final result is retrieved using the CalculatorResults accessibility ID.

The test includes basic setup for desired capabilities, opens a session to the local WinAppDriver server, performs the operation, asserts the correct result, and closes the session cleanly.The code was executed on a Windows 10 machine with Java 21, JUnit 4.13.2, Selenium 4.29.0, Appium Java Client 5.0.2, WinAppDriver 1.2.1 No additional runners or Serenity integration was used.

**Test File**: Question8Test.java

```
import org.junit.*;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.DesiredCapabilities;
import java.util.concurrent.TimeUnit;
import static org.junit.Assert.assertEquals;
import java.net.MalformedURLException;
import java.net.URI;
import io.appium.java_client.windows.WindowsDriver;
public class Question8Test {
   @Test
   public void AdditionTest() throws MalformedURLException
   {

      // capabilities
      DesiredCapabilities capabilities = new DesiredCapabilities();
       capabilities.setCapability("app",
                  "Microsoft.WindowsCalculator_8wekyb3d8bbwe!App"); // calc
       capabilities.setCapability("platformName",
                  "Windows"); // platform

       // driver opens session
       WindowsDriver<WebElement> driver =
                  new WindowsDriver<>(URI.create("http://127.0.0.1:4723")
                              .toURL(), capabilities);
       System.out.println("Windows Driver Session ID: "
                              + driver.getSessionId());
       driver.manage().timeouts().implicitlyWait(20,
                  TimeUnit.SECONDS);
```

```
        // 25 + 75 = 100
        driver.findElementByName("Two").click();
        driver.findElementByName("Five").click();       // 25
             driver.findElementByName("Plus").click();   // +
        driver.findElementByName("Seven").click();
        driver.findElementByName("Five").click();       // 75
             driver.findElementByName("Equals").click(); // =

        // 100 == 100
        String total = driver
                   .findElementByAccessibilityId("CalculatorResults")
                   .getText().replace("Display is", "").trim();
        assertEquals("100", total);

        driver.quit();
    }
}
```

**Question 9: API Automation with Rest Assured**
**Write a test using Rest Assured to send a GET request to a public API (e.g., https://jsonplaceholder.typicode.com/posts/1) and validate that the response status is 200 and the title field is not empty.**

The following test uses Rest Assured to validate a public API response, as requested. It sends a GET request and checks that the response status code is `200` and that the `title` field is not empty.

The test is implemented as a standalone Java JUnit test and can accept a custom endpoint as a system property, allowing it to be reused across different APIs without modifying the code.

```java
package com.ssc.Assessment.tests;
import org.junit.Test;
import static io.restassured.RestAssured.*;
import static org.hamcrest.Matchers.*;
public class Question9Test {
     @Test
   public void getRequestToAPublicAPI() {
       String url = System.getProperty("test.api.url",
               "https://jsonplaceholder.typicode.com/posts/1");
           given()
           .when()
               .get(url)
           .then()
               .statusCode(200)
               .body("title", not(emptyOrNullString()));
   }
}
```

Usage:

```
mvn clean verify
-Dtest.api.url=https://jsonplaceholder.typicode.com/posts/2
```

To enhance reporting and integration, I proceeded to change implementation of the same test using Serenity and SerenityRest.

**Test File**: Question9Test.java

```
package com.ssc.Assessment.tests;
import org.junit.Test;
import org.junit.runner.RunWith;
import net.serenitybdd.junit.runners.SerenityRunner;
//import static io.restassured.RestAssured.*;
import static net.serenitybdd.rest.SerenityRest.*;
import static org.hamcrest.Matchers.*;
@RunWith(SerenityRunner.class)
public class Question9Test {
     @Test
     public void getRequestToAPublicAPI() {
       String url = System.getProperty("test.api.url",
               "https://jsonplaceholder.typicode.com/posts/1");
           given()
           .when()
               .get(url)
           .then()
               .statusCode(200)
               .body("title", not(emptyOrNullString()));
   }
}
```

This version allows the test to be integrated into the Serenity Dashboard.

**Question 10: Test Metrics Dashboard**
**Describe and sketch (or outline) a simple test metrics dashboard you would create using GitHub Actions and test reports from Serenity. What metrics would you include and why?**

After completing Questions 3, 6, 7, and 9 of this assessment, I collected the related Java tests into a single Maven project. I then created a working CI pipeline using GitHub Actions and integrated it with Serenity to generate detailed test reports. The YAML workflow is triggered on every push to the main branch. It runs all tests, generates a Serenity report, and uploads it as an artifact.

To ensure that a full report is always generated, even if some tests fail, I included ***`continue-on-error: true.`*** After the test run, ***`mvn serenity:aggregate`*** builds the final HTML dashboard. The result is a downloadable artifact ***`serenity-report.zip`***, containing the complete test output.

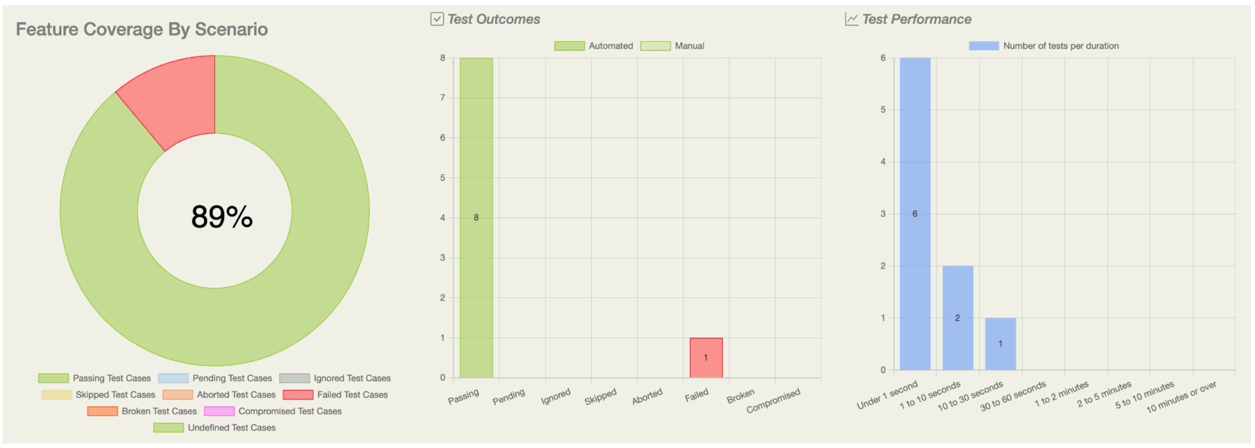
If I were to outline the ideal test metrics dashboard, it would look very similar to the default Serenity HTML report. I'd include:

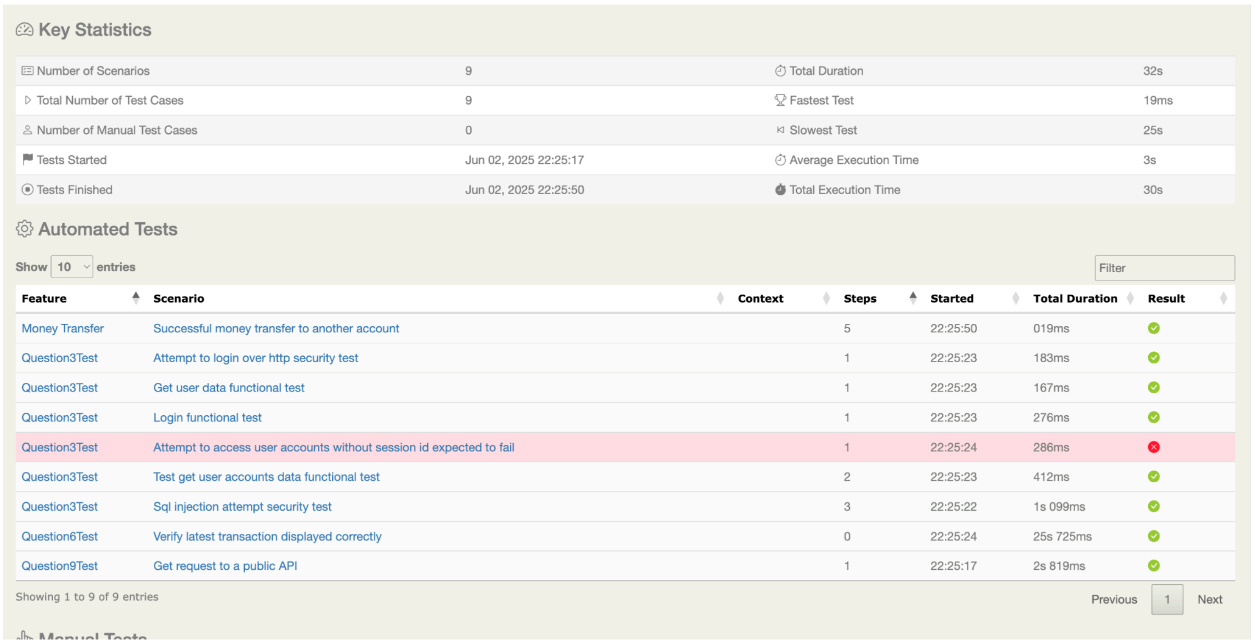
1. Total number of tests, to give an overview of coverage
2. Passed, failed, and skipped test counts, as a quick view of test health
3. Test duration metrics, to spot potential performance bottlenecks
4. Step-by-step test breakdowns, to help with root cause analysis
5. Tags and filters, for grouping and navigating tests by category
6. Detailed error messages with stack traces and screenshots for debugging

I would structure the dashboard simply: a summary panel at the top, followed by recent test results, and then expandable logs per scenario at the bottom.

The Serenity dashboard shown below is generated automatically after each GitHub Actions run. It provides a clear and navigable view of test results. Reports like this help QA teams and stakeholders quickly spot regressions or unstable areas.

**Screenshots:** [Serenity Report 1](), [Serenity Report 2]()





Key Statistics

| | | | |
|---|---|---|---|
| Number of Scenarios | 9 | Total Duration | 32s |
| Total Number of Test Cases | 9 | Fastest Test | 19ms |
| Number of Manual Test Cases | 0 | Slowest Test | 25s |
| Tests Started | Jun 02, 2025 22:25:17 | Average Execution Time | 3s |
| Tests Finished | Jun 02, 2025 22:25:50 | Total Execution Time | 30s |

Automated Tests

Show 10 entries Filter

| Feature | Scenario | Context | Steps | Started | Total Duration | Result |
|---|---|---|---|---|---|---|
| Money Transfer | Successful money transfer to another account | | 5 | 22:25:50 | 019ms | ✓ |
| Question3Test | Attempt to login over http security test | | 1 | 22:25:23 | 183ms | ✓ |
| Question3Test | Get user data functional test | | 1 | 22:25:23 | 167ms | ✓ |
| Question3Test | Login functional test | | 1 | 22:25:23 | 276ms | ✓ |
| Question3Test | Attempt to access user accounts without session id expected to fail | | 1 | 22:25:24 | 286ms | ✗ |
| Question3Test | Test get user accounts data functional test | | 2 | 22:25:23 | 412ms | ✓ |
| Question3Test | Sql injection attempt security test | | 3 | 22:25:22 | 1s 099ms | ✓ |
| Question6Test | Verify latest transaction displayed correctly | | 0 | 22:25:24 | 25s 725ms | ✓ |
| Question9Test | Get request to a public API | | 1 | 22:25:17 | 2s 819ms | ✓ |

Showing 1 to 9 of 9 entries

Previous 1 Next

The Serenity report shown above is also available as a downloadable artifact from GitHub Actions:

[Download serenity-report.zip]()

*Note: This artifact will remain available for **90 days** from the date of upload (until approximately August 31, 2025).*