**Real Time Monitoring of automation execution with Grafana using Playwright**

**INFLUX DB :**

InfluxDB is an open-source time-series database designed to handle high write and query loads. It is built to be scalable, resilient, and efficient for collecting, storing, querying, and visualizing time-series data. Here are some key features .

1. Time-Series Data Model: InfluxDB is optimized for time-series data, making it ideal for storing metrics, events, and other time-stamped data.

2. SQL-like Query Language: InfluxDB uses a SQL-like query language for querying data, making it easy to retrieve and analyze time-series data.

3. Retention Policies: InfluxDB allows you to define retention policies to automatically expire and delete old data, helping to manage storage space.

4. Tagging and Fielding: InfluxDB supports tagging and fielding of data points, allowing for efficient querying and filtering of time-series data.

**Setting up InfluxDB to Collect Test Results:**

To set up InfluxDB to collect results from your Playwright, TypeScript, and Cucumber tests, follow these steps:

1. Install InfluxDB:

Install InfluxDB by following the instructions provided on the official website: [InfluxDB Downloads] <https://www.influxdata.com/downloads/>

2. Start InfluxDB:

Start the InfluxDB service after installation. If you installed it using a package manager, you can typically start it with a command like:

sudo service influxdb start

for windows :

A screen shot of a computer screen

Description automatically generated

Run the influxdb on the localhost :

A screenshot of a computer

Description automatically generated

**INFLUX DB CLIENT :**

This is used for to check the how many metrices are collected or not it have small configuration :

* After running the influxdb on the localhost 8086 it asks for the sine up you give your username and password as you want and organization name and during this you need to give your bucket name after that a token will be generated .
* And after configuration on to the server we need to put credentials during connection with influxdb client

**Steps :**

Open cmd prompt on particular path where influx.exe visible

After going inside the particular folder path use below cmd after

**Influx**

If after above cmd help section of influxdb visible means you are now ready to connect

After that use below commad for credentials which you use sign up during on the influxdb server

161781@DHFTBK3 MINGW64 ~/Downloads/influxdb2-client-2.7.3-windows-amd64

$ ^Cflux config create --config-name cloudrnd --host-url "http://localhost:8086" --org "Einfochips=>" --token "SBjaOrUz2UsNvHHaQrqzfMMxy30Ew\_GUdFFwyiuETXj5uyf-zzq5I7sA24ZgDdC-JxYIctKJtWWhJCm2ZGXjgA==" --active

A screen shot of a computer

Description automatically generated

* Now we ready to start writing flux queries and get the real time bucket data use time stamp for to get particular data …

A black screen with white lines

Description automatically generated

5. Instrument Test Execution:

Within your Playwright, TypeScript, and Cucumber test framework, instrument your tests to collect relevant metrics and results during test execution. This may include metrics like test duration, pass/fail status, errors encountered, etc.

6. Send Results to InfluxDB:

Use InfluxDB's HTTP API to send test results from your test framework to InfluxDB. You can use libraries like `node-fetch` to make HTTP requests from your TypeScript code.

7. Analyze Results:

Once your test results are stored in InfluxDB, you can use InfluxDB's query language or other visualization tools to analyze the results, identify trends, and gain insights into your test execution.

Conclusion:

Integrating InfluxDB into your Playwright, TypeScript, and Cucumber test framework allows you to collect, store, and analyze test results in a scalable and efficient manner. By storing test results in a time-series database like InfluxDB, you can gain valuable insights into test performance, identify issues, and improve the overall quality of your software.

**GRAFANA:**

**1. Grafana**

Grafana is an open-source platform for monitoring and observability, allowing you to create customizable dashboards and visualizations for analyzing and understanding your data. It supports a wide range of data sources and provides powerful querying and visualization capabilities.

**2. Installation and Setup**

**2.1. Installation**

You can install Grafana on various operating systems, including Linux, Windows, and macOS. Here are general steps to install Grafana:

Windows and macOS:

Download the installer from the official Grafana website.

Use the username and password

Admin / admin

**2.2. Configuration**

After installation, you can configure Grafana by editing the grafana.ini configuration file. ( make changes in it if required)Common configurations include:

**Data Sources:** Configure data sources such as InfluxDB, Prometheus, MySQL, etc.

**Authentication:** Set up user authentication using LDAP, OAuth, or basic authentication. (changes if required)

**SMTP:** Configure SMTP settings for email notifications. (changes if required)

**SSL:** Enable SSL/TLS for secure connections. (changes if required)

Refer to the official Grafana documentation for detailed configuration options.

**2.3. Starting Grafana**

Start the Grafana server using the following command:

**bash**

sudo service grafana-server start

or directly go into the bin to start the server

Or on Windows, start Grafana from the Start menu or command prompt.

**3. Data Sources Configuration**

Grafana supports a wide range of data sources including InfluxDB, Prometheus, MySQL, PostgreSQL, etc. Here's how to configure a data source:

A screenshot of a computer

Description automatically generated

Log in to Grafana: Open your web browser and navigate to **http://localhost:3000.**

Log in with Default Credentials: The default username/password is **admin/admin**.

Add Data Source: Go to Configuration > Data Sources > Add Data Source.

Select Data Source Type: Choose the type of data source you want to configure (e.g., InfluxDB).

Configure Connection: Enter the connection details including URL, database name, authentication credentials, etc.

Save and Test: Click on "Save & Test" to verify the connection.

**4. Creating Dashboards**

Dashboards in Grafana allow you to visualize and analyze your data. Here's how to create a dashboard:

Log in to Grafana: Open your web browser and navigate to <http://localhost:3000>.

Create Dashboard: Go to Create > Dashboard > Add new panel.

Add Panels: Choose the type of visualization you want to add (e.g., Graph, Singlestat, Table).

Configure Panels: Configure the panel settings, queries, and visualization options.

Organize Panels: Arrange and resize panels as desired.

Save Dashboard: Click on Save dashboard and provide a name for your dashboard.

5. Visualization and Customization

Grafana provides extensive options for customizing and configuring visualizations. Some common customization options include:

Queries: Customize data queries using various query languages (e.g., PromQL, SQL).

Visualization Types: Choose from a wide range of visualization types including graphs, gauges, tables, etc.

Alerts: Set up alerts to be notified when certain conditions are met.

Annotations: Add annotations to mark events or significant points on graphs.

Templating: Use variables to dynamically change the data displayed in your dashboard.

6. Administration and Maintenance

Grafana offers features for administration and maintenance tasks including:

User Management: Manage users, roles, and permissions.

Backup and Restore: Backup and restore Grafana configurations and data.

Logging and Monitoring: Monitor Grafana server logs and performance metrics.

Upgrades: Keep Grafana up to date by installing the latest versions.

Refer to the official Grafana documentation for detailed information on administration and maintenance tasks.

Conclusion

Grafana is a powerful tool for monitoring and visualization, allowing you to create customized dashboards and analyze your data effectively. By following the steps outlined in this documentation, you can integrate Grafana into your system, configure data sources, create dashboards, and manage visualizations to gain valuable insights into your data.

**Integration of the playwright framework with influxdb and populate the automated scenarios results on to the Grafana :**

**Purpose :**

This integration allows you to capture and visualize performance metrics of web applications using Playwright, store the data in InfluxDB, and create dashboards in Grafana for monitoring and analysis.

**Prerequisites**

you have the following installed and set up before going on further:

Node.js and npm installed on your system.

**InfluxDB installed and running.(server) =>**

**A screenshot of a computer

Description automatically generated**

**Grafana installed and running.(server) =>**

**A screenshot of a computer

Description automatically generated**

npm install playwright (already done if you are using framework playwright typescript and cucumber)

**2. Install InfluxDB Node.js Client**

Install the InfluxDB Node.js client to interact with InfluxDB:

JavaScript client library to write data in InfluxDB Cloud Serverless

**npm install @influxdata/influxdb-client**

**npm install @influxdata/influxdb-client@latest**

**A screen shot of a computer

Description automatically generated**

Below command for InfluxDB management APIs allow you to manage your InfluxDB instances, including creating, deleting, and querying databases, as well as managing users and permissions =>

**npm install --save @influxdata/influxdb-client-apis**

**3. Set Up Grafana**

Set up Grafana and configure it to connect to InfluxDB as a data source .

A screenshot of a computer

Description automatically generated

Follow the below link : and no forget to select the query language as the flux during input credential if you are using influx server

<https://docs.influxdata.com/influxdb/cloud/tools/grafana/>

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**4. Create Playwright Script** : (prefer to select the **hooks.ts** where your browser start and stop configurations)

Now Write a Playwright script to perform the desired actions on your web application and capture performance metrics.

Ex:

A screenshot of a computer program

Description automatically generated

below is the code of my hooks.ts | main changes are in before , beforeAll , after and afterall

Check and see what changes need to done all remains constant in the hooks.ts | **make only necessary changes here highlighted portion are changed if some of the things you already configured than just ignored**

import { BeforeAll, AfterAll, Before, After, Status } from "@cucumber/cucumber";

import { APIRequestContext, Browser, BrowserContext, request } from "@playwright/test";

import { fixture } from "./pageFixture";

import { invokeBrowser } from "../helper/browsers/browserManager";

import { getEnv } from "../helper/env/env";

import { createLogger } from "winston";

import RESTRequest from "../../src/helper/playwright/API/RESTRequest";

import { options } from "../helper/util/logger";

import { InfluxDB, Point } from '@influxdata/influxdb-client';

import \* as osUtils from 'os-utils';

import \* as os from 'os';

import { promisify } from 'util';

const sleep = promisify(setTimeout);

let influxdbWriter: any; // Type for InfluxDB Writer

let browser: Browser;

**let apiContext: APIRequestContext;**

**let context: BrowserContext;**

**let currentScenarioName: string = '';**

**let startTime: number | undefined;**

**let endTime: number | undefined;**

**let TotalPassed: number = 0;**

**let TotalFailed: number = 0;**

**let TotalUndefined: number = 0;**

**let TotalSkipped: number = 0;**

**let CountScenarios: number = 0;**

BeforeAll(async function () {

try {

getEnv();

// invoke browser using playwright

browser = await invokeBrowser();

**// influx db configuratons** | these things you choose during run of the influxdb server (on localhost)

**const token = '<your-token>';**

**const org = '<your organization name>';**

**const bucket = '<your bucket name>';**

**const url = 'http://localhost:8086';**

**const influx = new InfluxDB({ url, token });**

**influxdbWriter = influx.getWriteApi(org, bucket);**

**} catch (error) {**

**console.error('Error in BeforeAll:', error);**

**}**

**});**

**Before(async function ({ pickle }) {**

**currentScenarioName = pickle.name + pickle.id;**

**startTime = Date.now();**

**});**

**After(async function ({ pickle, result }) {**

**const scenarioName = currentScenarioName;**

**let statusTag = 'failed';**

**if (result?.status === Status.PASSED) {**

**statusTag = 'passed';**

**TotalPassed++;**

**} else if (result?.status === Status.FAILED) {**

**statusTag = 'failed';**

**TotalFailed++;**

**} else if (result?.status === Status.UNDEFINED){**

**statusTag = 'undefined';**

**TotalUndefined++;**

**} else if (result?.status === Status.SKIPPED){**

**statusTag = 'skipped'**

**TotalSkipped++;**

**}**

**endTime = Date.now();**

**const duration = endTime - startTime;**

**if (statusTag === 'passed') {**

**const point = new Point('DM\_Pro')**

**.tag('scenario', scenarioName)**

**.tag('status', statusTag)**

**.intField('total\_passed', TotalPassed)**

// note : you can add more tags and points as you want to captures like duration and other error and etc.

**influxdbWriter.writePoint(point);**

**} else if (statusTag === 'failed') {**

**const point = new Point('DM\_Pro')**

**.tag('scenario', scenarioName)**

**.tag('status', statusTag)**

**.intField('total\_failed', TotalFailed)**

**influxdbWriter.writePoint(point);**

**} else if (statusTag === 'undefined') {**

**const point = new Point('DM\_Pro')**

**.tag('scenario', scenarioName)**

**.tag('status', statusTag)**

**.intField('total\_undefined', TotalUndefined)**

**influxdbWriter.writePoint(point);**

**} else if (statusTag === 'skipped') {**

**const point = new Point('DM\_Pro')**

**.tag('scenario', scenarioName)**

**.tag('status', statusTag)**

**.intField('total\_skipped', TotalSkipped)**

**influxdbWriter.writePoint(point);**

**}**

**});**

**AfterAll(async function () {**

**try {**

**await influxdbWriter.close();**

**await browser.close();**

**} catch (error) {**

**console.error('Error in AfterAll:', error);**

**}**

**});**

**// Before hook for scenarios tagged with @API**

**Before("@API", async function ({ pickle }) {**

**console.log("This is API Test");**

**// Set up API context and RESTRequest**

**const scenarioName = pickle.name + pickle.id;**

**apiContext = await request.newContext(); // Initialize apiContext**

**this.rest = new RESTRequest(apiContext);**

**fixture.logger = createLogger(options(scenarioName));**

**});**

**// Before hook for scenarios tagged with @UI**

**Before("@UI", async function ({ pickle }) {**

**const scenarioName = pickle.name + pickle.id;**

**// Create a new browser context for UI tests**

**context = await browser.newContext({**

**recordVideo: {**

**dir: "test-results/videos",**

**},**

**});**

**// Create a new page and set viewport size**

**const page = await context.newPage();**

**fixture.page = page;**

**await fixture.page.setViewportSize({ width: 1310, height: 700 });**

**fixture.logger = createLogger(options(scenarioName));**

**});**

// Function to get storage state based on the user

function getStorageState(user: string): string | { cookies: { name: string; value: string; domain: string; path: string; expires: number; httpOnly: boolean; secure: boolean; sameSite: "Strict" | "Lax" | "None"; }[]; origins: { origin: string; localStorage: { name: string; value: string; }[]; }[]; } {

if (user.endsWith("admin")) return "src/helper/auth/admin.json";

else if (user.endsWith("lead")) return "src/helper/auth/lead.json";

}

**START AUTOMATION :**

**A screen shot of a computer

Description automatically generated**

**Automation Results Metrices Collected in real time on influxdb server :**

**A screenshot of a computer

Description automatically generated**

**Check metrices into the influxdb client : check the collected data into database**

**Steps :**

1. Install influxdb cli from below link

<https://docs.influxdata.com/influxdb/cloud/reference/cli/influx/>

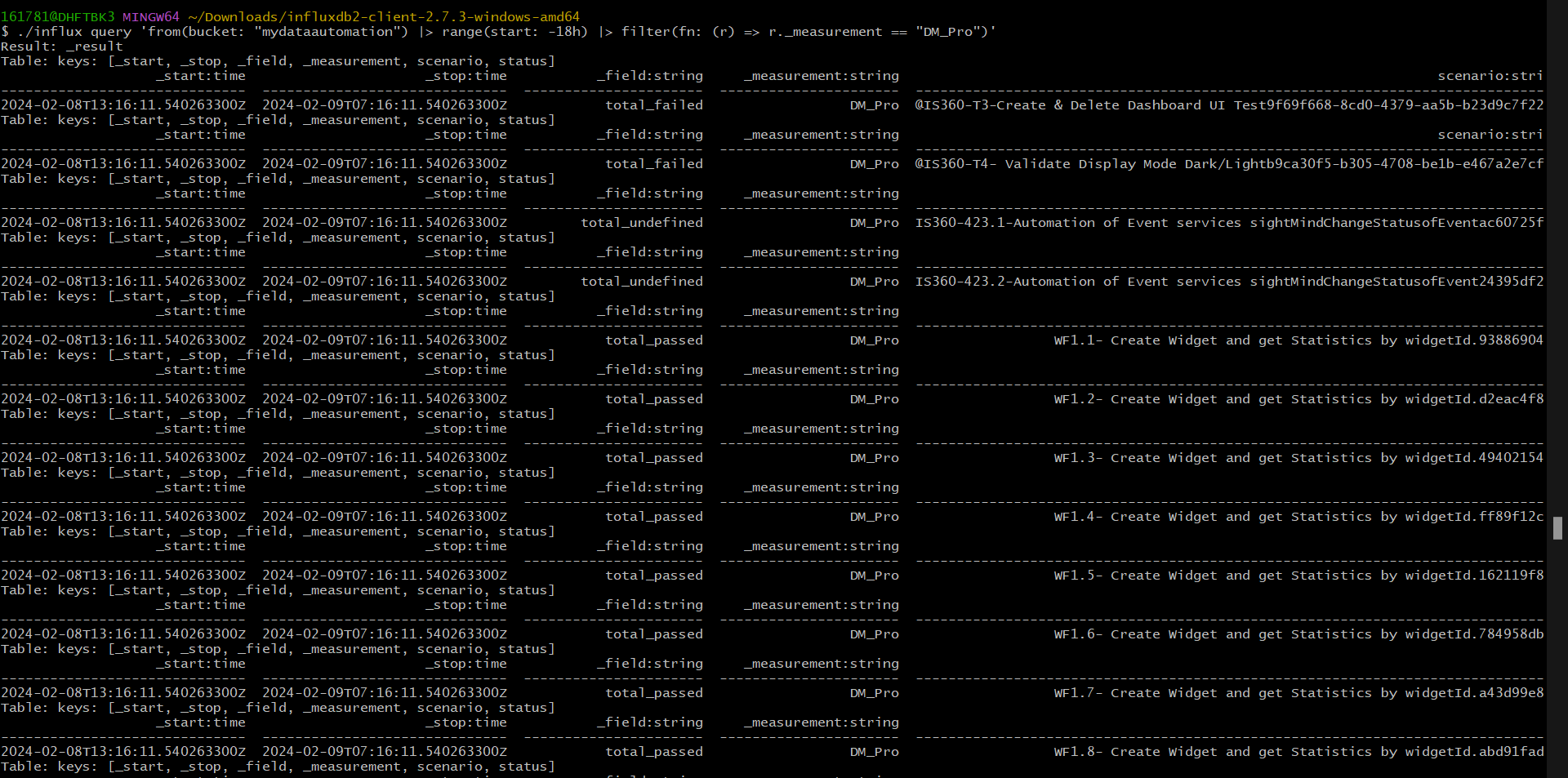
1. open cmd and go to influxdb cli path and
2. type influx if below interface means influx db cli is ready to setup

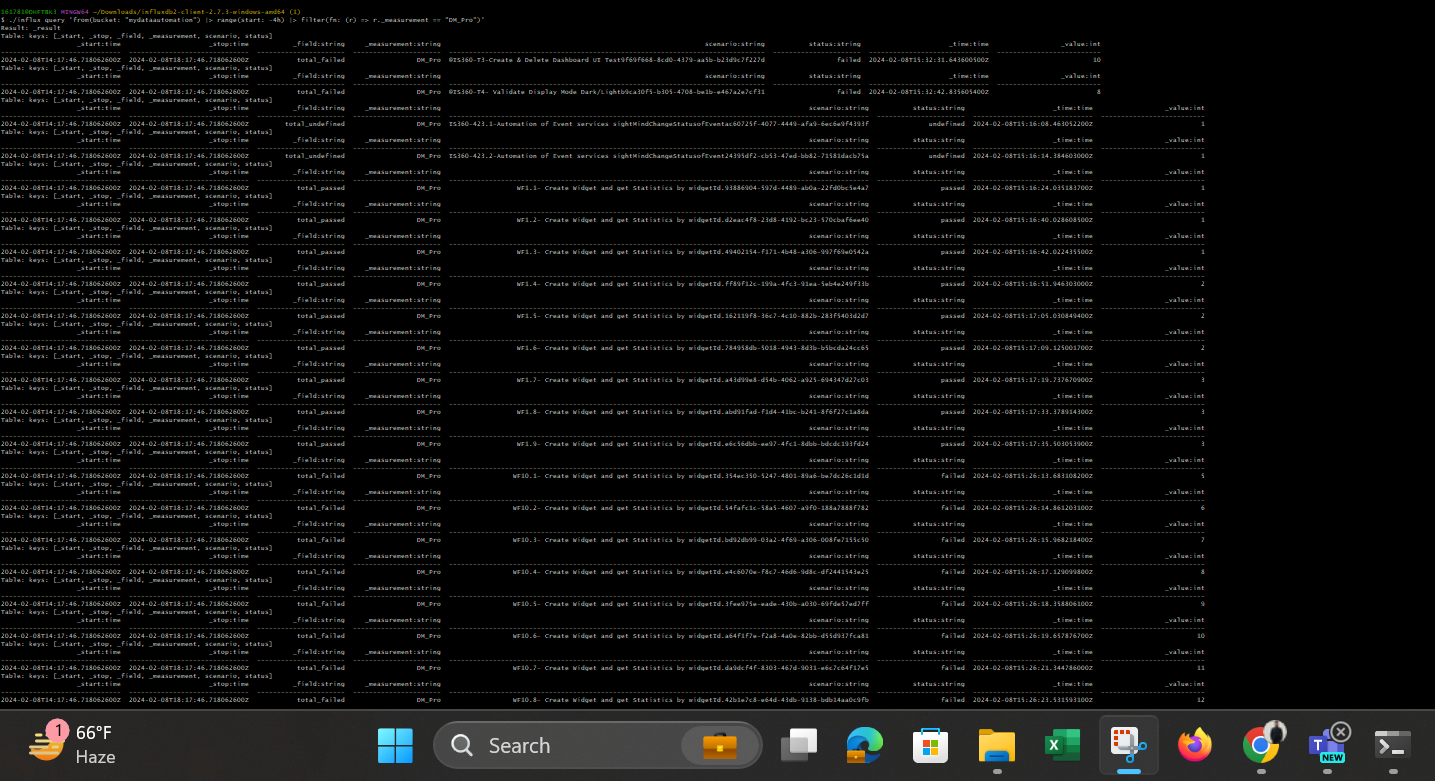
\Users\161781\Downloads\influxdb2-client-2.7.3-windows-amd64 (1)> influx config create --config-name cloudrnd --host-url "http://localhost:8086" --org "Einfochips=>" --token "SBjaOrUz2UsNvHHaQrqzfMMxy30Ew\_GUdFFwyiuETXj5uyf-zzq5I7sA24ZgDdC-JxYIctKJtWWhJCm2ZGXjgA==" --active

1. do start querying :

for to see data inside particular bucket and what data is collected

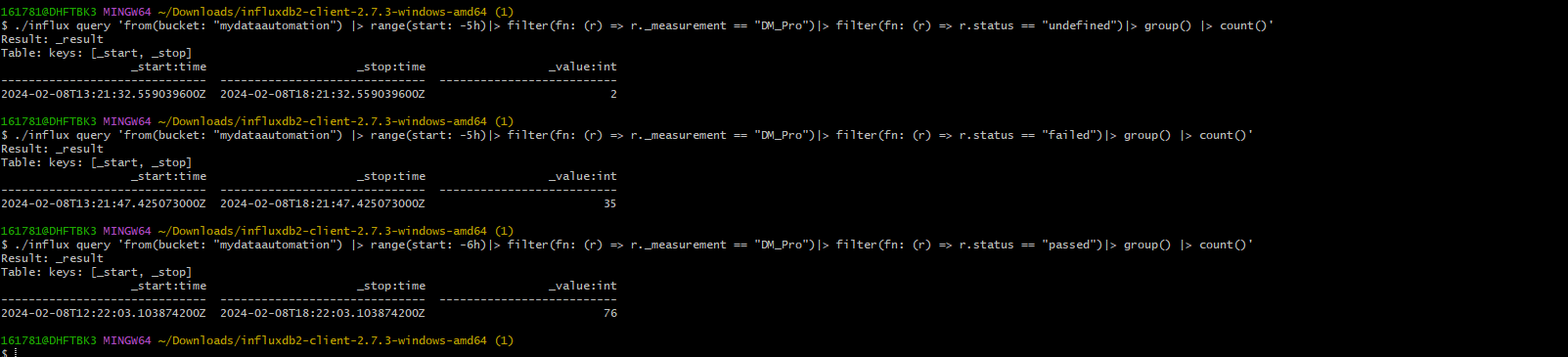
influx query 'from(bucket: "mydataautomation") |> range(start: -4h) |> filter(fn: (r) => r.\_measurement == "DM\_Pro")'





For to see passed , failed, skipped and undefined results scenarios

influx query 'from(bucket: "mydataautomation") |> range(start: -45m)|> filter(fn: (r) => r.\_measurement == "DM\_Pro")|> filter(fn: (r) => r.status == "failed")|> group() |> count()'

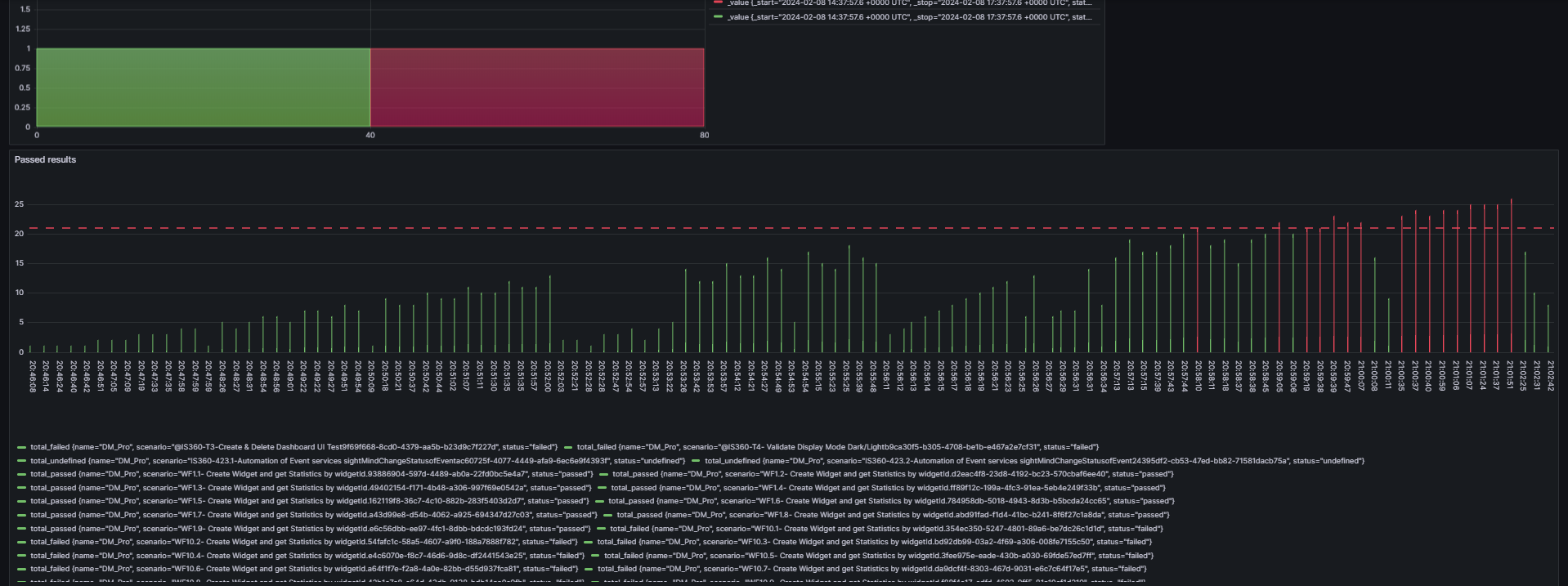


**And more …**

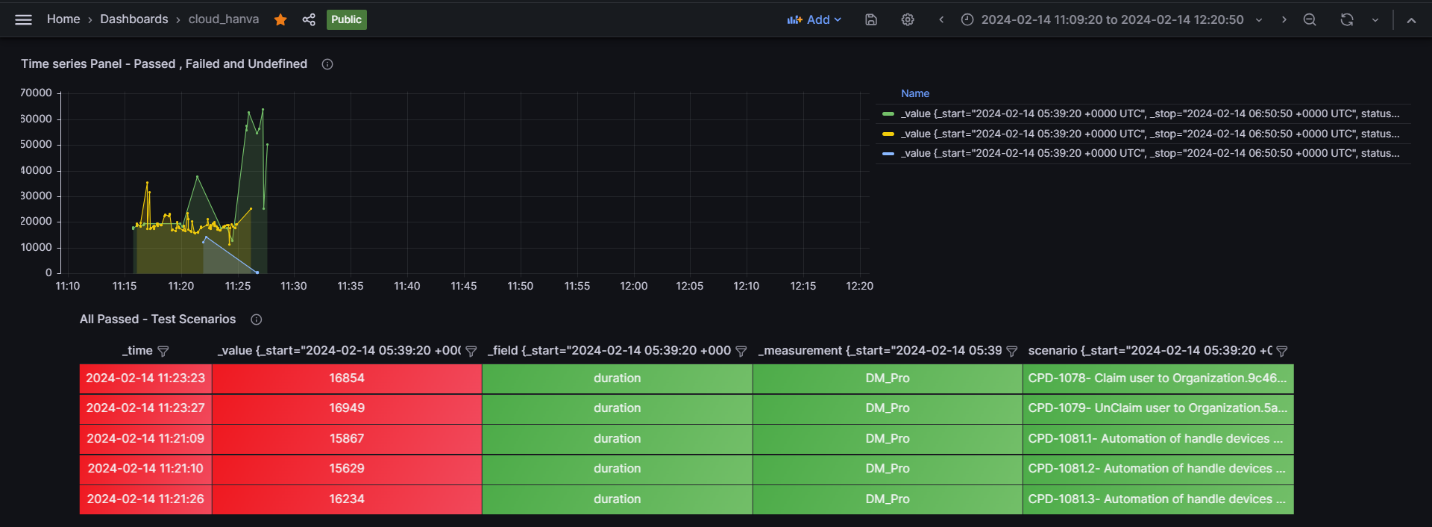
**Grafana output :**

**A screenshot of a video game

Description automatically generated**

****

**And other**

**A screenshot of a computer

Description automatically generated**

**Conclusion**

By following these steps, you can automate the integration between Playwright, InfluxDB, and Grafana for performance testing and monitoring of your web applications. This allows you to efficiently capture, store, and visualize performance metrics, enabling better insights into your application's behavior and performance over time.

**PROS :**

1. Comprehensive Monitoring: Provides detailed insights into performance metrics.
2. Visualization: Powerful visualization capabilities for test results.
3. Historical Analysis: Enables historical analysis of test data over time.
4. Alerting: Allows proactive monitoring and notification of issues.
5. Scalability: Can handle large volumes of data and scale as needed.

**Cons:**

1. Complexity: Adds complexity to setup and maintenance.
2. Resource Overhead: Requires additional system resources.
3. Learning Curve: Requires familiarity with new technologies.
4. Maintenance: Requires ongoing management and updates.
5. Dependency Management: Introduces additional dependencies and potential points of failure.