Testprojekt 3

# Rahmenbedingungen

* **Abgabetermin:**

30.06.2024, 12:00 Uhr.

* **Umfang der Abgabe:**
* Meldung per E-Mail an [olivia.macolic@dsgf.de](mailto:olivia.macolic@dsgf.de), danach sind keine Anpassungen am Portfolio vorgesehen.
* 30 Min Präsentation der Ergebnisse mit Fragerunde
* **Rückfragen:** 
  + Rückfragen per E-Mail an [olivia.macolic@dsgf.de](mailto:olivia.macolic@dsgf.de) senden.
  + Je nach Bedarf wird ein gemeinsamer Termin eingestellt.

# Aufgabe

**Automation** und **Dokumentation** von **Einkaufprozessen im Online-Shop "Swag Labs"** mittels relevanter Werkzeuge. Diese Aufgabe führt durch alle für die Aufgabe relevanten Werkzeuge.

## **Schritt: Für die Aufgabe müssen folgende Vorbereitungen getroffen werden:**

* Installation von Python, Robotframework und Selenium (und anderer Bibliotheken falls nötig) **Ѵ**
* Installation von VSCode **Ѵ**
* Einrichtung von Geckodriver **Ѵ**
* Installation und Einrichtung von Docker oder Podman **Ѵ**
* Aufsetzen einer SQL-Datenbank in Docker oder Podman inklusive Webinterface **Ѵ**

## **Schritt: Erstelle mit Robotframework folgenden Test:**

* Besuche die Seite <https://www.saucedemo.com/> **Ѵ**
* Führe alle möglichen Login Kombinationen durch **Ѵ**
* Führe mit jedem User einen Kauf durch **Ѵ**
* Nach erfolgreichem Kauf sollte der Produktname, sowie der Preis des Produktes und der entsprechende User in der SQL-Datenbank gespeichert werden

## **Hinweise zur Umsetzung:**

* Sollte ein Einkauf nicht erfolgreich sein, sollte dies ebenfalls in der Tabelle ersichtlich sein
* Schreibe die entsprechenden Keywords um Daten in der SQL-Datenbank zu speichern selber in Python (Nicht die vorhandene Database Library dafür verwenden)
* Verwende für den Robotframework Test den Datadriven Ansatz
* Verwende zum interagieren mit dem Browser die SeleniumLibrary
* Dokumentiere deine Schritte in einem Git Repository

# Ergebnisse

* Robotframework Bericht und LOG im Git
* Sourcefiles im Git
* Dokumentation über Installation und Konfigurationsschritte der verwendeten Tools im Git
* Präsentation in Teams

**Robot Framework – Wichtige Grundlagen & Struktur**

**🐍 Python**

* Grundlage für Robot Framework – läuft komplett in Python.
* Installation über [python.org](https://python.org) und Paketmanager pip.

**🤖 Robot Framework**

* Das eigentliche Testframework für automatisierte Tests.
* Installation: pip install robotframework
* Arbeitet mit speziellen .robot-Dateien für Tests.

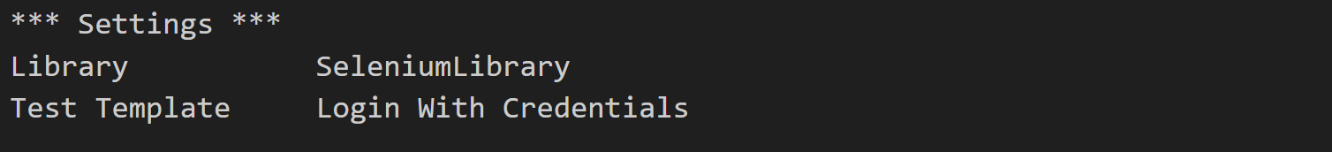
**🌐 SeleniumLibrary**

* Robot Framework Library, um echte Browser zu steuern (z. B. Firefox, Chrome).
* Ermöglicht Aktionen wie:
  + Elemente klicken
  + Texte eingeben
  + Seiteninhalte prüfen

**🔧 Geckodriver**

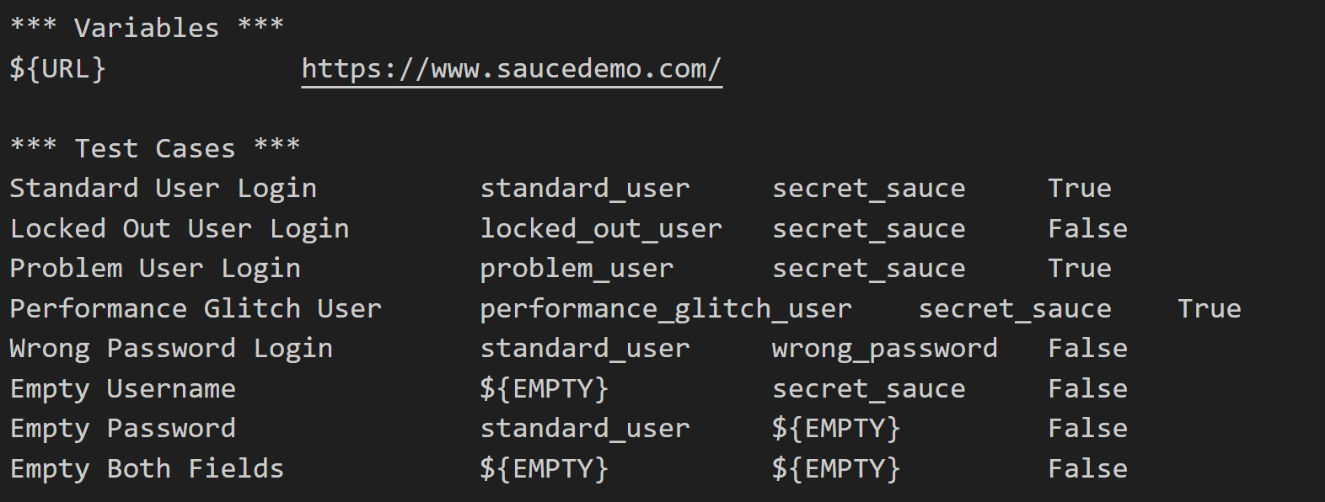
* „Brücke“ zwischen SeleniumLibrary und Firefox.
* Steuert Firefox über Selenium.
* Muss separat heruntergeladen und im Systempfad oder an bekannter Stelle liegen.

**Aufbau einer Robot Framework Datei (.robot)**

****

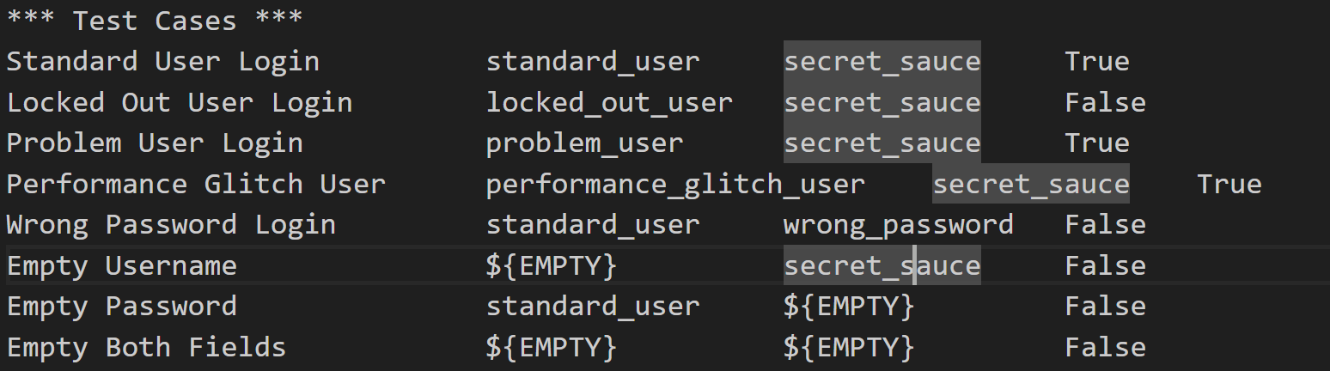
Lädt die benötigten Libraries.

Definiert „Test Templates“, damit Testfälle eine Vorlage (Keyword) gemeinsam nutzen.



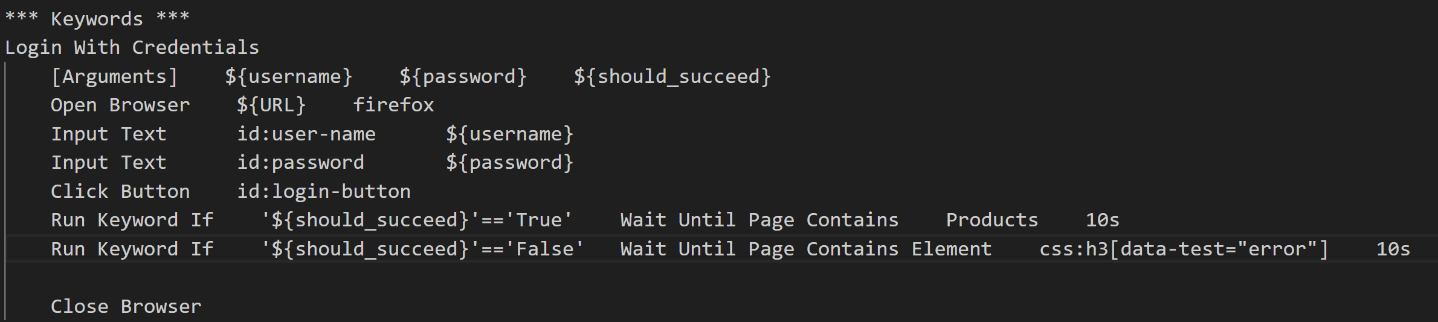
Definiert Variablen für wiederverwendbare Werte (z. B. URLs).

Sauberer, weil Änderungen zentral möglich sind.



Hier stehen die einzelnen Testfälle.

Jede Zeile ruft ein Keyword (Test Template) mit Parametern auf, z. B. Benutzername, Passwort, erwartetes Ergebnis.



Hier sind wiederverwendbare Befehle definiert (ähnlich wie Funktionen).

In diesem Beispiel:

Browser öffnen, Login-Daten eingeben, Login ausführen

Prüfen, ob Login erfolgreich war oder Fehlermeldung angezeigt wird

Browser schließen

**Ablauf bei Testausführung**

Robot Framework parst die .robot-Datei.

Erkennt alle Testfälle unter \*\*\* Test Cases \*\*\*.

Führt jeden Testfall nacheinander aus.

Nutzt die Anweisungen aus \*\*\* Keywords \*\*\*.

Erstellt am Ende folgende Reports:

output.xml (technisch, für CI/CD)

log.html (Detaillierte Ansicht jedes Schritts)

report.html (Übersicht der Testergebnisse)

Demo-Webseite & Zugangsdaten

https://www.saucedemo.com/ ist eine öffentliche Demo-Website für UI-Tests.

**Bekannte Benutzeraccounts:**

|  |  |  |
| --- | --- | --- |
| **Benutzername** | **Passwort** | **Verhalten** |
| standard\_user | secret\_sauce | Erfolgreicher Login |
| locked\_out\_user | secret\_sauce | Login schlägt fehl (gesperrt) |
| problem\_user | secret\_sauce | Login ok, UI zeigt Fehler (visuelle Bugs) |
| performance\_glitch\_user | secret\_sauce | Login ok, aber sehr langsam |
| error\_user | secret\_sauce | Ungewöhnliche Bild-Fehler |
| visual\_user | secret\_sauce | optional, manchmal deaktiviert |

**Robot Framework Output Dateien**

**1. output.xml**

**Was ist das?**

* Die **Haupt-Ausgabedatei** im XML-Format.
* Enthält alle Informationen über den Testlauf, inklusive aller Schritte, Status, Zeitstempel, Fehler, Logs, Metadaten.

**Inhalt**

* Jede Test-Suite, jeder Testfall, jedes Keyword (Schritt) wird hier als XML-Element mit Attributen (z. B. Status, Dauer) gespeichert.
* Detailinformationen zu Fehlern, Warnungen, Logs.
* Eignet sich perfekt zur automatischen Weiterverarbeitung.

**Wie liest du sie?**

* Am besten mit speziellen Tools oder Robot Framework eigenen Werkzeugen:
  + z. B. rebot-Tool:

rebot output.xml  
erzeugt log.html und report.html aus der XML-Datei.

* Du kannst auch XML-Viewer verwenden, aber das ist meist unpraktisch.

**Wofür wird sie genutzt?**

* **CI/CD-Pipelines** (Jenkins, GitHub Actions, GitLab CI, etc.) lesen output.xml ein, um automatisiert Testergebnisse auszuwerten.
* Andere Tools können daraus Berichte generieren oder in Dashboards integrieren.
* Grundlage für log.html und report.html.

**2. log.html**

**Was ist das?**

* Detaillierter **Testlauf-Report** in HTML.
* Sehr gut für manuelle Analyse.
* Zeigt alle Schritte, Testfälle, Status und Zeitdauer.

**Inhalt**

* Übersicht aller Tests mit Status (Pass, Fail, Skip).
* Für jeden Testfall:
  + Detaillierte Schritte (Keywords)
  + Ausgeführte Befehle (z. B. Klicks, Texteingaben)
  + Erfolg/Misserfolg einzelner Schritte
  + Zeitstempel
  + Fehlermeldungen
* (Optional) Screenshots bei Fehlern, wenn aktiviert.
* Filtermöglichkeiten (z. B. nur Fehler anzeigen).

**Wie liest du sie?**

* Einfach im Browser öffnen.
* Interaktive Navigation durch Testlauf.
* Kann zum Debuggen genutzt werden:
  + Wo genau ist ein Test fehlgeschlagen?
  + Welche Schritte wurden ausgeführt?
  + Welche Fehlermeldung gab es?

**Wofür wird sie genutzt?**

* Entwickler, Tester und QA-Teams verwenden die Datei, um Testdurchläufe zu analysieren.
* Wichtig für Fehlersuche und Verbesserung der Tests.
* Im Team teilen oder archivieren.

**3. report.html**

**Was ist das?**

* Kompakte **Übersicht** aller Testergebnisse.
* Fokus auf Gesamtstatus, keine Schritt-für-Schritt-Details.

**Inhalt**

* Gesamtanzahl der Tests
* Anzahl der bestandenen / fehlgeschlagenen / übersprungenen Tests
* Gesamtlaufzeit
* Zusammenfassung auf Suite- und Testfall-Ebene

**Wie liest du sie?**

* Öffne die Datei im Browser.
* Sie zeigt auf einen Blick, wie erfolgreich der Testlauf war.
* Du siehst schnell, ob z. B. alle Tests durchgelaufen sind.

**Wofür wird sie genutzt?**

* Schneller Überblick für Manager, Stakeholder oder automatisierte Systeme.
* Meist erstes Ergebnis nach einem Testlauf, um zu entscheiden, ob alles OK ist.
* Kombiniert mit log.html für detaillierte Analysen.

**Typischer Workflow im Einsatz**

1. Du führst Tests lokal oder in der CI/CD-Umgebung mit robot tests.robot aus.
2. Robot Framework erzeugt output.xml.
3. Aus output.xml werden automatisch log.html und report.html generiert.
4. Du öffnest report.html, um eine schnelle Übersicht zu bekommen.
5. Bei Problemen öffnest du log.html, um den Fehler und den genauen Schritt zu sehen.
6. Für automatisierte Abläufe (z. B. Jenkins) wird output.xml eingelesen und ausgewertet, um Build-Ergebnisse anzuzeigen oder Benachrichtigungen zu triggern.

**🛒 Ablauf eines Einkaufs auf saucedemo.com:**

1. Login
2. Produktübersicht → "Add to cart" klicken
3. Klick auf Warenkorb (oben rechts)
4. Klick auf "Checkout"
5. Eingabe von Vorname, Nachname, PLZ
6. Klick auf "Continue"
7. Klick auf "Finish"
8. Bestätigung sehen (z. B. „THANK YOU FOR YOUR ORDER“)

Noch nicht umgesetzt, da komplizierter als erwartet, ggf später wenn noch zeit:

**🔧 Empfehlung für dein Projekt:**

1. Dokumentiere in deiner README.md:

Für parallele Testausführung: pip install robotframework-pabot und das Skript run\_tests.ps1 verwenden.

**🐳 Option A: Docker (empfohlen, wenn du keine besonderen Gründe für Podman hast)**

**🔧 1. Docker Desktop installieren (Windows)**

* Lade Docker Desktop von der offiziellen Seite herunter:  
  👉 https://www.docker.com/products/docker-desktop
* Installiere es wie ein normales Windows-Programm.
* Starte Docker Desktop.
* Es sollte nun automatisch im Hintergrund laufen (Symbol in der Taskleiste)

Herunterladen: Windows-Subsystem für Linux 2.5.7 warum war diese installation nötig?

Gute Frage! Die Installation des Windows-Subsystems für Linux (WSL) — in deinem Fall WSL Version 2.5.7 — war nötig, weil:

🧠 Docker Desktop (unter Windows) intern Linux braucht

Docker läuft nativ unter Linux, weil es auf Linux-Kernel-Funktionen wie Namespaces und cgroups angewiesen ist.

Da Windows keinen Linux-Kernel hat, braucht Docker unter Windows einen „Trick“:

→ WSL2 ist dieser Trick. Es stellt eine echte Linux-Umgebung innerhalb von Windows bereit.

🔍 Warum genau WSL 2?

WSL2 (im Gegensatz zu WSL1) bringt einen vollwertigen Linux-Kernel mit, der es Docker ermöglicht, ohne zusätzliche VM oder Hyper-V direkt zu laufen.

Kurz gesagt:

Docker benötigt Linux.

Windows hat kein Linux.

Also installiert Docker WSL2, um intern Linux laufen zu lassen.

⚙️ Das passiert bei der Installation:

Docker erkennt, dass WSL2 nicht (oder nicht aktuell genug) installiert ist.

Es lädt automatisch die passende Version herunter.

Optional wird auch eine Linux-Distribution (z. B. Ubuntu) vorgeschlagen/installiert, aber das ist nicht zwingend nötig für Docker.

Wenn du also gefragt hast, warum das installiert wurde:

👉 Damit Docker unter Windows überhaupt funktionieren kann.

**✅ Notwendige Schritte zur Einrichtung von Docker auf Windows (mit WSL2):**

1. **Installation von Docker Desktop für Windows**  
   – Das Hauptprogramm, das den Docker-Client und den Docker-Daemon bereitstellt.
2. **Installation des Windows-Subsystems für Linux (WSL2)**  
   – Docker Desktop verwendet WSL2 als **Backend**, um Container unter Windows auszuführen.  
   – Deshalb wurde **„Windows-Subsystem für Linux 2.5.7“** installiert.  
   → Ohne WSL2 funktioniert Docker unter Windows **nicht korrekt**.
3. **Einrichtung/Start von Docker Desktop**  
   – Beim ersten Start richtet sich Docker selbstständig ein, startet WSL2-Integrationen und startet den Daemon.

**🧪 Nur zum Testen, ob Docker korrekt funktioniert:**

1. **docker --version**  
   → Zeigt, dass der Docker-Client installiert ist.
2. **docker run hello-world**  
   → Startet einen Testcontainer, der überprüft, ob:
   * Docker mit dem Daemon kommunizieren kann,
   * Images vom Docker Hub heruntergeladen werden können,
   * Container erstellt und ausgeführt werden können.

**📦 1. Projektordner vorbereiten**

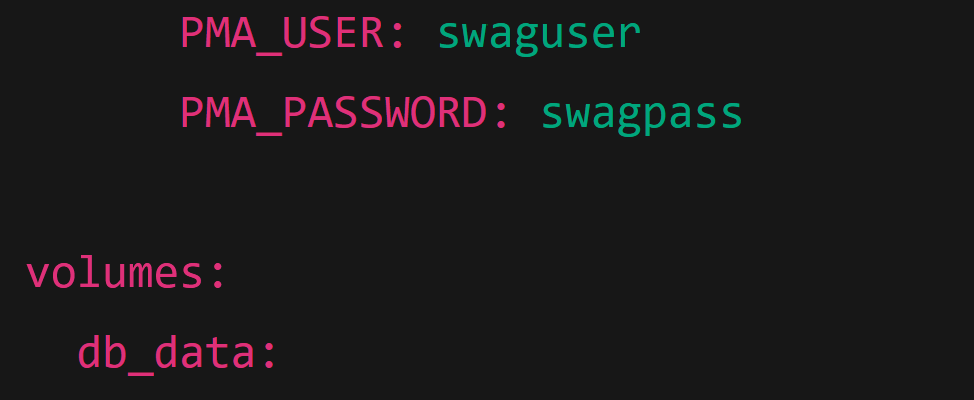
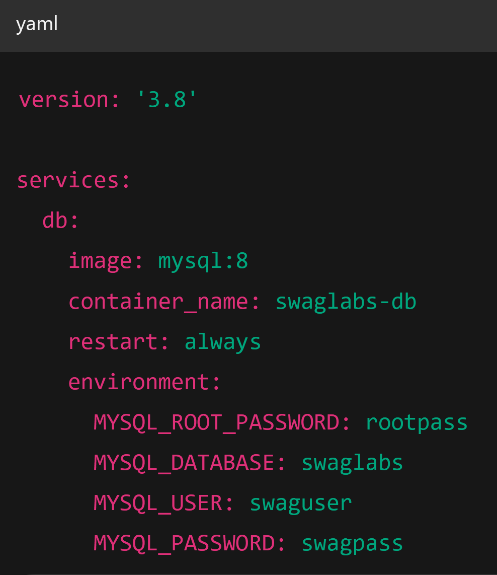
Lege in deinem Projekt einen neuen Ordner an:

Ein Bild, das Text, Schrift, Screenshot, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

**📝 2. docker-compose.yml erstellen**

Erstelle eine Datei docker-compose.yml mit folgendem Inhalt:



Erklärung der yml:

**version: '3.8'**

* Gibt die Version des Docker Compose Dateiformats an.
* Version 3.8 ist eine relativ aktuelle Version mit guten Features.
* Hinweis: Ab Version 3 ist die Versionsangabe optional, aber sie hilft, die Kompatibilität sicherzustellen.

**services:**

* Definiert die einzelnen Container, die zusammen in deinem Setup laufen.

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

**image: mysql:8**

* Nutzt das offizielle MySQL Image in Version 8 aus Docker Hub.

**container\_name: swaglabs-db**

* Gibt dem Container einen festen Namen, damit du ihn leichter findest.

**restart: always**

* Der Container wird automatisch neu gestartet, falls er abstürzt oder Docker neu startet.

**environment:**

* Hier werden Umgebungsvariablen an den MySQL-Container übergeben:
  + MYSQL\_ROOT\_PASSWORD: Passwort für den MySQL Root-User (wichtig!)
  + MYSQL\_DATABASE: Name der automatisch erstellten Datenbank (swaglabs)
  + MYSQL\_USER und MYSQL\_PASSWORD: Ein neuer User mit diesen Zugangsdaten wird erstellt und erhält Zugriff auf die genannte Datenbank.

**ports:**

* "3306:3306" bedeutet, dass Port 3306 des Hosts (dein Rechner) auf Port 3306 des Containers weitergeleitet wird.
* Dadurch kannst du lokal oder aus anderen Containern auf MySQL zugreifen.

**volumes:**

* db\_data:/var/lib/mysql
* Ein Docker Volume namens db\_data wird auf den Datenordner von MySQL im Container gemountet.
* So bleiben deine Daten persistent, auch wenn der Container gelöscht wird.

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

**image: phpmyadmin/phpmyadmin**

* Offizielles phpMyAdmin Image für die Verwaltung von MySQL über Webbrowser.

**container\_name: swaglabs-pma**

* Fester Containername.

**restart: always**

* Container wird bei Absturz/Neustart automatisch wieder hochgefahren.

**ports:**

* "8080:80" bedeutet: Port 8080 deines Hosts wird auf Port 80 des Containers geleitet.
* Du kannst phpMyAdmin dann im Browser über http://localhost:8080 aufrufen.

**environment:**

* PMA\_HOST: db
  + phpMyAdmin soll sich mit dem MySQL-Container db verbinden (über das interne Docker-Netzwerk).
* PMA\_USER und PMA\_PASSWORD: Zugangsdaten für den MySQL-User, mit dem sich phpMyAdmin verbinden soll.

Ein Bild, das Schrift, Text, Grafiken, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

Hier definierst du das Volume db\_data, das im MySQL-Container genutzt wird.

Volumes sind persistent und überleben Container-Neuerstellungen.

So bleiben deine Daten erhalten.

bei volumen ist ja zb nichts eingetragen. 1. warum und 2. muss das dann überhaupt da stehen

eine **Deklaration** eines benannten Volumes ohne weitere Optionen oder Pfadangaben.

Das bedeutet:

* **db\_data ist ein Docker-Volume, das Docker selbst verwaltet.**
* Docker legt an einem Standard-Ort auf deinem Rechner einen Ordner für dieses Volume an und kümmert sich um die Verwaltung.
* Du musst keinen Pfad oder weitere Details angeben, weil es ein *managed* Volume ist.
* Das Volume wird dann in deinem Container an /var/lib/mysql gemountet, damit MySQL seine Daten persistent speichern kann.

**2. Muss das Volume überhaupt deklariert werden?**

* **Ja, wenn du ein benanntes Volume wie db\_data in services.db.volumes verwendest, musst du es unter volumes: deklarieren.**
* Wenn du es nicht deklarierst, erzeugt Docker Compose beim Starten automatisch ein anonymes Volume, das du nicht einfach wiederverwenden kannst.
* Durch die Deklaration kannst du das Volume gezielt benennen und verwalten.
* Falls du stattdessen einen **Bind-Mount** mit einem lokalen Pfad nutzen willst (z.B. ./mysql-data:/var/lib/mysql), dann brauchst du das Volume nicht zu deklarieren, weil du ja einen festen Ordner auf deinem Rechner direkt angibst.

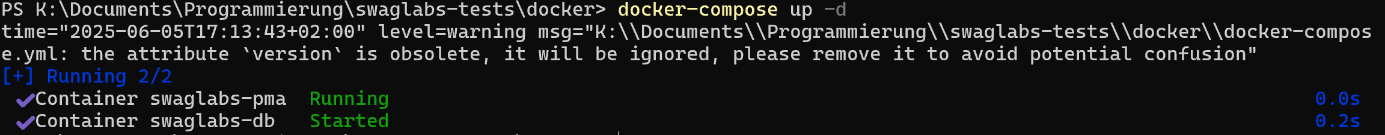
**▶️ 3. Container starten**

Öffne eine PowerShell in deinem Projektverzeichnis und führe aus:

Ein Bild, das Text, Schrift, Screenshot, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

In meinem Fall natürlich cd K:\Documents\Programmierung\swaglabs-tests\docker\



Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

KI-generierte Inhalte können fehlerhaft sein.

Erstellen der benötigten Tabelle für die geforderten Grunddaten Username,Produktname und Preis:  
CREATE TABLE purchases (

id INT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) NOT NULL,

product\_name VARCHAR(100) NOT NULL,

price DECIMAL(10,2) NOT NULL,

purchase\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

MyDatabase.py erstellt:

Ein Bild, das Text, Screenshot, Software, Betriebssystem enthält.

KI-generierte Inhalte können fehlerhaft sein.  
1 Importiert die pymysql-Bibliothek, die wir nutzen, um in Python eine Verbindung zu einer MySQL-Datenbank herzustellen und SQL-Befehle auszuführen.

4 init\_\_ ist der Konstruktor der Klasse, der automatisch aufgerufen wird, wenn du ein Objekt von MyDatabase erzeugst.

5 self.connection = None bedeutet, dass wir anfangs keine Datenbank-Verbindung haben. self.connection ist ein Attribut des Objekts, das später die Datenbank-Verbindung speichert.

8 Die geöffnete Verbindung speichern wir in self.connection, damit wir später darauf zugreifen können und pymysql.connect(...) öffnet die Verbindung zur MySQL-Datenbank mit diesen Angaben.

14 cursorclass=pymysql.cursors.DictCursor sorgt dafür, dass Ergebnisse von Abfragen als Wörterbücher (dicts) zurückgegeben werden, das ist meist leichter zu verarbeiten als reine Tupel.

17 execute\_sql ist eine Methode, die einen SQL-Befehl ausführt.

18 Zuerst wird geprüft, ob self.connection existiert (also ob wir verbunden sind). Wenn nicht, werfen wir einen Fehler.

20 with self.connection.cursor() as cursor: öffnet einen Cursor, über den wir SQL ausführen können. Der Cursor wird automatisch geschlossen, wenn der Block endet.

21 cursor.execute(sql\_query) führt die übergebene SQL-Anweisung aus (z.B. ein INSERT oder SELECT).

22 self.connection.commit() bestätigt die Änderung in der Datenbank. Das ist wichtig bei Befehlen, die Daten verändern (INSERT, UPDATE, DELETE).

25 Wenn self.connection gesetzt ist (also verbunden),

26 wird self.connection.close() aufgerufen, um die Verbindung zu beenden.

27 Danach wird self.connection wieder auf None gesetzt, damit der Zustand klar ist.