RR – QA Automation Assignment

Description

Thank you for showing interest in Rapyuta Robotics and for wanting to be a Rapyutian.
This assignment showcases my skills in **UI and API test automation** using Python, Selenium, and Pytest.
The target demo website is: TMDB Discover, a movie and TV show listing platform for testing purposes.

Assignment Scope

1. **Filtering Options**
2. **Categories:** Popular, Trending, Newest, Top Rated
3. **Titles:** Movie or TV show titles
4. **Type:** Movies or TV Shows
5. **Year of Release:** Using start and end year
6. **Rating:** Star rating filter
7. **Genre:** E.g., Action
8. **Pagination**
9. Navigate through pages
10. Validate movie titles load correctly
11. **Negative / Known Issues**
12. Refreshing/accessing pages with specific slugs may fail
13. Last few pagination pages may not function

Test Strategy

1. Step-by-step Test Descriptions

Test Case	Steps	Expected Result
Category Filter	1. Open Home Page 2. Click on a category 3. Wait for titles to load	URL contains category slug, movie titles displayed
Type Filter	1. Open Type dropdown 2. Select Movie/ TV Show 3. Validate selection	Dropdown shows selected type

Test Case	Steps	Expected Result
Year Range Filter	1. Select start and end year 2. Validate UI shows selected range	Displayed movies are within the year range
Star Rating Filter	1. Select a star rating 2. Validate selected star	Movies are filtered based on rating
Genre Filter	1. Open Genre dropdown 2. Select Action 3. Validate UI	Dropdown shows selected genre, movies filtered
Pagination	1. Wait for pagination 2. Click next page 3. Verify page number	Movies for next page displayed correctly
Category Page Refresh	1. Open category URL 2. Refresh page 3. Validate movie titles	Titles reload successfully
Broken Pages Check	1. Navigate last 3 pages 2. Verify movie titles	No missing titles

---

2. Functional Test Suite Implementation

Technologies / Libraries Used: - **Python** – Test scripts
- **Selenium** – Web UI automation
- **Pytest** – Test runner
- **pytest-html** – HTML report generation with screenshots
- **Requests** – API testing
- **webdriver-manager** – Browser driver management

Features Implemented: - Fully automated tests for **UI filters** and **pagination**
- **API tests** for categories, rating, year range, and pagination
- **Logging** implemented with `logging` module (info, step, error logs)
- Screenshots captured **on failures** and attached to HTML report
- Configurable test data stored in `utils/test_data.py`
- Config file (`utils/config.py`) for base URL, waits, browser, and paths

Test Execution Command Example:

```
pytest --html=reports/report.html --self-contained-html
```

---

3. Logging

- Logs stored in **console** and optionally to file
- Logs include:
- Test start/end

• Step execution
• Selected filters
• API responses
• Errors / assertion failures

Example Log:

```
INFO     tests.test_homepage: Selecting category → Popular
INFO     tests.test_homepage: Titles loaded successfully
INFO     tests.test_homepage: Current URL: /popular
ERROR    tests.test_homepage: Category Filter Test Failed: AssertionError:
Expected '/popular' in URL
```

---

4. Screenshots & HTML Reports

• Screenshots are saved in `reports/screenshots` on test failure

• Pytest-html embeds screenshots in the report automatically

• Reports contain:

• Test status (Pass/Fail)
• Steps and logs
• Screenshot attachments for failed tests

---

5. API Test Implementation

• **Endpoints Tested:**
• Category listing (`popular`, `top_rated`, `now_playing`)
• Rating filter (vote_average ≤ 5)
• Year range filter
• Pagination

API Test Example:

```
response = requests.get(f"{BASE_URL}/{category}?page=1&api_key={API_KEY}")
assert response.status_code == 200
data = response.json()
assert "results" in data
```

• Validates:
• Status code

- Expected JSON keys
- Data correctness (title, release_date, vote_average, id)

---

6. Defects Found

- Refreshing category URLs sometimes fails
- Pagination works for first few pages; last few pages may not display movies

---

7. CI/CD Integration Approach

Although not implemented, the approach would be:

1. **GitHub Actions / Azure DevOps** pipeline
2. Steps:
3. Install Python dependencies (`requirements.txt`)
4. Run Pytest tests
5. Generate HTML report and save artifacts
6. Optionally upload screenshots for failed tests
7. Schedule **daily or on PR** runs to catch regression issues

Pipeline Example (GitHub Actions YAML snippet):

```
name: Run UI/API Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: '3.11'
      - run: pip install -r requirements.txt
      - run: pytest --html=reports/report.html --self-contained-html
      - uses: actions/upload-artifact@v3
        with:
          name: Test-Report
          path: reports/
```

---

8. Folder Structure

```
rr-qa-automation-assignment/
│
├── pages/
│   ├── base_page.py
│   └── home_page.py
│
├── tests/
│   ├── test_ui_filters.py
│   ├── test_api_endpoints.py
│   └── conftest.py
│
├── utils/
│   ├── config.py
│   ├── test_data.py
│   └── logger.py
│
├── reports/
│   └── screenshots/
│
├── requirements.txt
├── pytest.ini
└── README.md
```

This document provides a **complete overview** of the test strategy, implementation, logging, reporting, defects, and CI/CD approach for submission on GitHub.