Role-Based UI WebApp Tester

Project Documentation

Table of Contents

1. Project Overview

2. System Architecture

3. Key Components

4. Installation and Setup

5. Usage Guide

6. Technical Details

7. Development Roadmap

8. Troubleshooting

1. Project Overview

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

The Role-Based UI WebApp Tester is an automated testing tool designed to crawl web applications, capture screenshots, generate Playwright tests using a role-based template approach, execute those tests, and produce comprehensive reports with screenshots and results. This tool is particularly useful for quickly assessing the functionality of UI elements across a web application without requiring manual test script creation.

Key Features:

- Automated web crawling and screenshot capture

- Role-based element detection and test generation

- Optimized HTML processing for faster analysis

- Robust selector generation based on element attributes

- Comprehensive test execution with visual mode support

- Detailed HTML reporting with element-level insights

2. System Architecture

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

The system follows a modular architecture with the following workflow:

1. Web Crawling: The crawler module navigates through the web application, capturing screenshots and HTML content.

2. Test Generation: The template generator analyzes the HTML content, identifies UI elements by their roles, and generates Playwright test scripts.

3. Test Execution: The test executor runs the generated tests, tracking the success or failure of interactions with each element.

4. Reporting: The reporter generates a comprehensive HTML report showing test results, element statistics, and screenshots.

3. Key Components

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

The project consists of several key Python modules, each responsible for a specific part of the workflow:

a) main.py

- Entry point for the application

- Orchestrates the entire testing process

- Handles command-line arguments

b) crawler.py

- Responsible for crawling the web application

- Uses Playwright to navigate and capture screenshots

- Saves HTML content for analysis

- Implements throttling and retry logic

c) template\_generator.py

- Analyzes HTML content to identify UI elements

- Uses a role-based approach to categorize elements

- Generates appropriate test actions based on element roles

- Creates robust selectors for reliable element identification

d) test\_executor.py

- Executes the generated test scripts

- Supports both headless and visual testing modes

- Captures test results and screenshots

e) element\_tracker.py

- Tracks the effectiveness of UI elements during testing

- Records whether elements are working properly based on expected behavior

- Provides detailed element-level insights

f) reporter.py

- Generates comprehensive HTML reports

- Displays statistics on element testing

- Shows before/after screenshots for each element

- Provides detailed test output

g) \_fallback\_detection.py

- Implements more aggressive element detection for pages where standard detection finds few elements

- Tries to find any potentially interactive elements on the page

4. Installation and Setup

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

Prerequisites:

- Python 3.7 or higher

- pip (Python package manager)

Installation Steps:

1. Clone or download the project repository

2. Install dependencies:

```bash

pip install -r requirements.txt

playwright install

```

Dependencies:

- Playwright: For browser automation and screenshot capture

- Requests: For HTTP requests

- BeautifulSoup4: For HTML parsing and analysis

- Jinja2: For HTML report template rendering

- Pillow: For image processing

5. Usage Guide

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

Basic Usage:

```bash

python main.py <base\_url>

```

Example:

```bash

python main.py https://master.d2l4isn05opwbv.amplifyapp.com/

```

Command-line Options:

- `--visual` or `-v`: Run tests in visual mode (browser will be visible)

Output Directories:

- Screenshots: `screenshots/`

- Generated tests: `generated\_tests/`

- Test results: `test\_results/`

- HTML report: `report.html`

6. Technical Details

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

Role-Based Element Detection:

The system uses a role-based approach to identify UI elements. Each element is categorized based on its HTML attributes, tag name, and other characteristics. The roles include:

- Button: Elements that can be clicked to trigger an action

- Link: Elements that navigate to another page or section

- Input: Text fields, textareas, and other input elements

- Checkbox: Toggle elements that can be checked/unchecked

- Radio: Radio button elements for selecting one option from a group

- Select: Dropdown menus and other selection elements

- Tab: Tab navigation elements

- Menu: Menu items and dropdown elements

- Dialog: Modal dialogs and popup windows

- Alert: Notification and alert elements

- Form: Form elements that can be submitted

- Slider: Range input elements

- Toggle: Switch elements

- Datepicker: Date input elements

- File: File upload elements

- Interactive: Any other interactive elements

Selector Generation:

The system generates robust selectors for each element based on its attributes:

1. ID-based selectors (most reliable)

2. Data attribute selectors (data-testid, data-cy, etc.)

3. Attribute-based selectors (name, type, etc.)

4. Class-based selectors

5. Text-based selectors

6. Position-based selectors (as a last resort)

Test Generation:

Tests are generated using templates for different actions:

- Click: For buttons, links, and other clickable elements

- Fill: For input fields and textareas

- Check: For checkboxes and radio buttons

- Select Option: For dropdown menus

- Detect: For elements that should be visible but not interacted with

- Submit: For form elements

- Slide: For slider elements

- Toggle: For toggle switches

7. Troubleshooting

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

Common Issues:

1. Element Not Found

- Problem: Test fails because an element cannot be found

- Solution: Check if the element is dynamically loaded or requires specific conditions to be visible

2. Navigation Timeout

- Problem: Test times out during navigation

- Solution: Increase the timeout value or check network connectivity

3. Selector Issues

- Problem: Element selectors are not reliable

- Solution: Use more specific selectors or add custom selectors for problematic elements

4. Memory Issues

- Problem: Process runs out of memory when testing large sites

- Solution: Limit crawl depth or implement pagination in the crawling process

5. Browser Crashes

- Problem: Browser crashes during testing

- Solution: Update Playwright or run with different browser options

For additional support or to report issues, please contact the development team.