AI-Powered Test Automation Suite Documentation

Table of Contents

Project Overview

Purpose

This is an end-to-end automated testing suite that combines AI-powered test case generation with browser automation. It transforms website analysis into comprehensive test documentation and executable test cases.

Key Features

- **Intelligent Website Scraping**: Extracts navigation, forms, buttons, and content structure
- **AI-Powered Test Generation**: Uses LLM agents to create user stories, test plans, and test cases
- **Automated Browser Testing**: Converts test cases into executable browser automation
- **Professional Documentation**: Generates Excel reports in industry-standard formats
- **Guest User Focus**: Specialized for testing public-facing website functionality

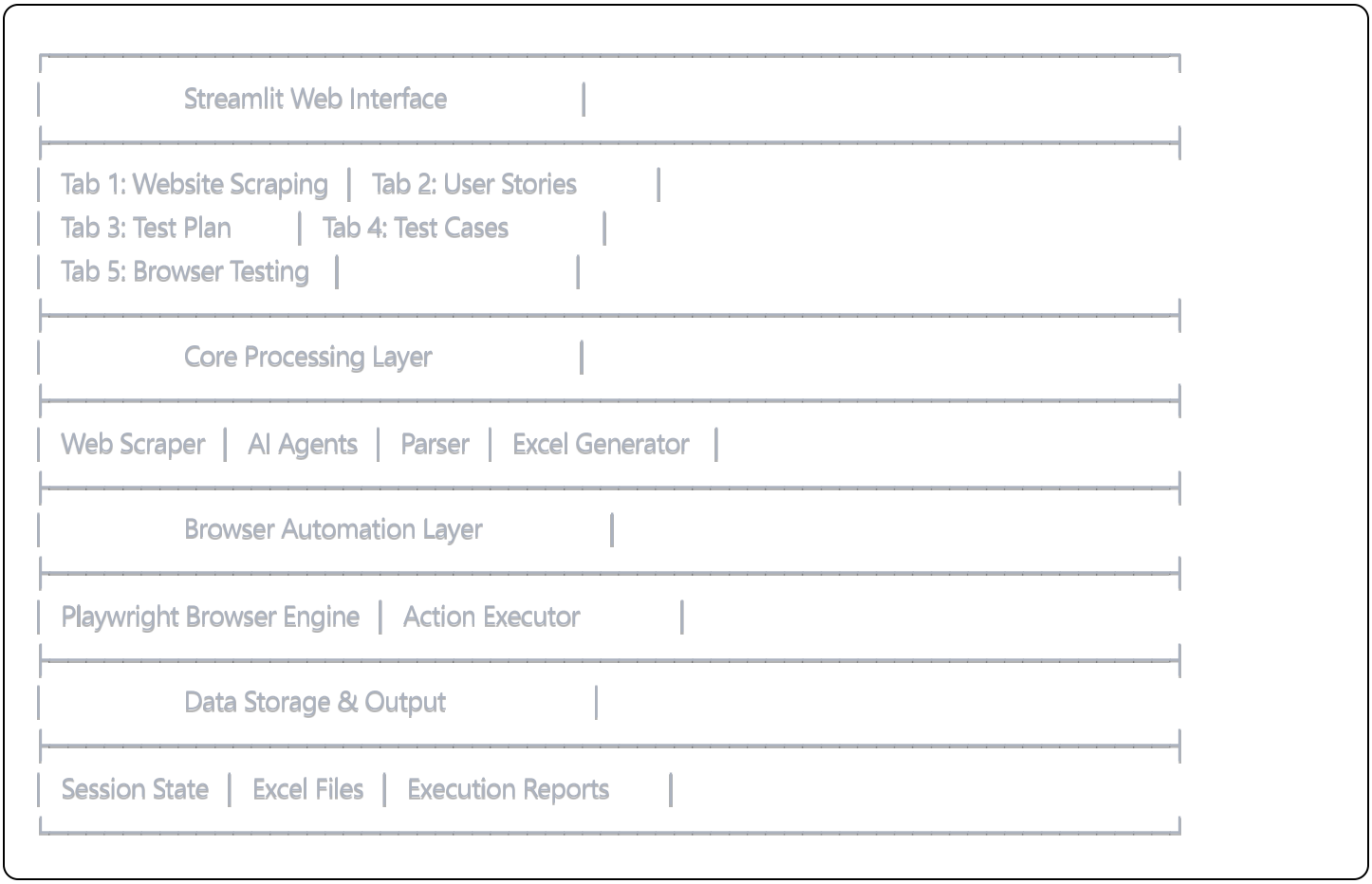
Technology Stack

- **Frontend**: Streamlit (Web UI)

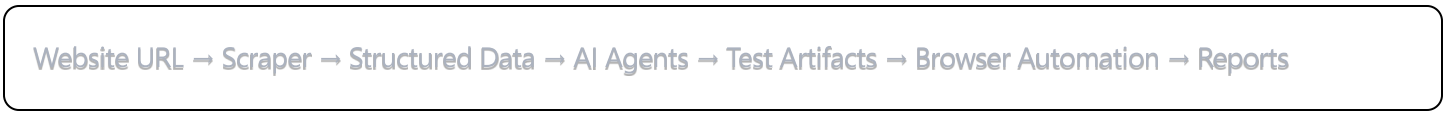
- **Web Scraping**: Playwright, BeautifulSoup
- **AI/LLM**: LangChain with Groq (Llama-3 model)
- **Browser Automation**: Playwright (async)
- **Document Generation**: OpenPyXL
- **Data Processing**: Python, JSON, Regex

Architecture

System Components



Data Flow



System Requirements

Hardware Requirements

- **RAM**: Minimum 4GB, Recommended 8GB+
- **Storage**: 2GB free space
- **Network**: Stable internet connection for AI API calls

Software Requirements

- **Python**: 3.8 or higher
- **Operating System**: Windows 10+, macOS 10.15+, or Linux Ubuntu 18+
- **Browser**: Chrome/Chromium (automatically managed by Playwright)

Required Python Packages

```
streamlit>=1.28.0
playwright>=1.40.0
beautifulsoup4>=4.12.0
langchain-groq>=0.1.0
langchain-core>=0.2.0
openpyxl>=3.1.0
lxml>=4.9.0
```

Installation & Setup

Step 1: Environment Setup

```bash
# Create virtual environment
python -m venv test_automation_env

# Activate environment
# Windows:
test_automation_env\Scripts\activate
# macOS/Linux:
source test_automation_env/bin/activate

# Install dependencies
pip install streamlit playwright beautifulsoup4 langchain-groq langchain-core openpyxl lxml
```

Step 2: Playwright Setup

```bash
# Install Playwright browsers
playwright install chromium
```

Step 3: API Configuration

```python
# Update GROQ_API_KEY in the code
GROQ_API_KEY = "your_groq_api_key_here"
```

Step 4: Launch Application

```bash
streamlit run main4.py
```

Component Details

1. Website Scraper Module

File: `scrape_website_sync()`

Purpose: Extracts structured data from websites

Key Functions:

- Navigation menu extraction
- Form field identification
- Button and link discovery
- Meta information parsing
- Alert/error message detection

Output Structure:

```json
```

```
{
  "navigation": [{"text": "Home", "href": "/"}],
  "page_info": {"title": "Site Title", "description": "..."},
  "forms": [{"action": "/submit", "fields": [...]}],
  "extra_buttons": [{"text": "Click Me", "type": "button"}],
  "interactive_links": [{"text": "Learn More", "href": "/about"}],
  "alert_messages": ["Error message"],
  "url": "https://example.com"
}
```


2. AI Agent System

LLM Provider: Groq (Llama-3 Model) **Framework**: LangChain

Agent Types:

1. **User Story Agent**: Generates Agile user stories from website features
2. **Test Plan Agent**: Creates formal test plan documentation
3. **Test Case Agent**: Develops detailed test cases with steps
4. **Test Data Agent**: Generates realistic test data for form inputs

Prompt Engineering:

- Context-aware prompts tailored for each agent type
- Structured output formatting
- Domain-specific terminology and best practices

3. Browser Automation Engine

Framework: Playwright (Async)

Key Features:

- Headless and headed browser modes
- Cross-platform compatibility
- Advanced selector strategies
- Error handling and timeouts
- Screenshot capabilities

Action Types:

- `open_website`: Navigate to URL
- `click_element`: Click buttons/links
- `fill_input`: Enter form data
- `verify_text`: Validate page content
- `verify_title`: Check page titles
- `verify_url`: Confirm navigation

4. Document Generation System

Library: OpenPyXL

Features:

- Professional Excel formatting
- Multiple worksheet support
- Custom styling (colors, fonts, alignment)
- Data validation
- Export functionality

Step-by-Step Workflow

Phase 1: Website Analysis

1. **Input URL**: User enters target website URL
2. **Scraping Process**:
   - Launch headless browser
   - Navigate to website
   - Extract DOM structure
   - Parse HTML elements
   - Identify interactive components
   - Store structured data

Phase 2: AI-Powered Content Generation

Step 1: User Story Generation

- **Input**: Scraped website structure

- **Process**: AI agent analyzes features and creates user stories
- **Output**: Numbered list of "As a Guest User" stories
- **Format**: `"As a Guest, I want to [action] so that [benefit]"`

Step 2: Test Plan Creation

- **Input**: Website structure analysis
- **Process**: AI creates formal test plan document
- **Output**: Professional test plan with scope, objectives, limitations

Step 3: Test Case Development

- **Input**: Website features + User stories
- **Process**:
  - Generate step-by-step test cases
  - Create positive and negative test scenarios
  - Add preconditions and expected results
  - Generate test data for inputs
- **Output**: Structured test case matrix

Phase 3: Test Automation

Step 1: Test Case Conversion

- **Input**: Structured test cases from Phase 2
- **Process**:
  - Parse natural language test steps
  - Map actions to browser commands
  - Create selector strategies
  - Define verification points
- **Output**: Executable browser automation scripts

Step 2: Browser Test Execution

- **Process**:
  - Launch browser instance
  - Execute test steps sequentially
  - Capture results and screenshots

- Handle errors and timeouts
- Generate execution reports

Step 3: Navigation Testing

- **Process**:
  - Test all navigation menu items
  - Verify page loads and content
  - Check URL changes
  - Validate expected content

Technical Implementation

Core Classes and Functions

Website Scraping

```python
def scrape_website_sync(url):
    """
    Synchronous website scraping using Playwright
    Returns structured data dictionary
    """
```

AI Agent System

```python
def call_llm_agent(prompt_template, input_key, input_value):
    """
    Generic LLM agent caller using LangChain
    """


def generate_user_stories(site_data):
def generate_test_plan(site_data):
def generate_test_cases(site_data):
def generate_test_data_for_case(testcase_string):
```

Browser Automation

```python
async def setup_browser():
    """Setup Playwright browser instance"""

async def execute_action(page, action_data):
    """Execute browser action with error handling"""

def convert_test_case_to_browser_actions(test_case, base_url, scraped_data):
    """Convert natural language test to browser commands"""
```

Data Processing Pipeline

1. Input Validation

- URL format verification
- Accessibility checks
- Error handling

2. Data Extraction

- DOM parsing with BeautifulSoup
- Element classification
- Relationship mapping

3. AI Processing

- Prompt construction
- API communication
- Response parsing

4. Output Generation

- Excel formatting
- Data structure conversion
- File export

Output Formats

1. User Stories Excel

Columns:

- ID (US-001, US-002, etc.)
- User Story (Full text)

Styling:

- Blue header background
- Auto-wrapped text
- Professional formatting

2. Test Cases Excel

Columns:

- Test Case ID
- User Story (Reference)
- Test Scenario
- Preconditions
- Test Steps
- Test Data
- Expected Result
- Actual Result
- Status
- Priority

3. Test Execution Report Excel

Columns:

- Test Case ID
- Test Scenario
- Test Steps (Executed)
- Expected Result
- Actual Result

- Status (PASS/FAIL/ERROR)
- Execution Timestamp

4. Button Test Report Excel

Columns:

- Button Label
- Clickable (Boolean)
- Loaded Successfully (Boolean)
- Errors (Text)
- Selector Used

Configuration Guide

AI Model Configuration

```python
# Groq API Settings
GROQ_API_KEY = "your_api_key"
MODEL_NAME = "meta-llama/llama-4-scout-17b-16e-instruct"

# LLM Parameters
temperature = 0.08  # Low for consistent results
max_tokens = 2048   # Sufficient for detailed responses
```

Browser Configuration

```python
# Playwright Browser Settings
browser_args = [
    "--no-sandbox",
    "--disable-gpu",
    "--window-size=1920,1080"
]

# Timeout Settings
page_timeout = 15000     # 15 seconds for page loads
element_timeout = 5000   # 5 seconds for element waits
```

Streamlit Configuration

```python
# Page Configuration
st.set_page_config(
    page_title="AI Test Automation Suite",
    layout="wide"
)

# Session State Management
# Uses st.session_state for data persistence across tabs
```

Troubleshooting

Common Issues

1. Browser Launch Failures

Symptoms: "Browser setup failed" error **Solutions**:

- Install Playwright browsers: `playwright install`
- Check system permissions
- Verify Chrome/Chromium availability

2. AI API Errors

Symptoms: LLM agent failures, timeout errors **Solutions**:

- Verify Groq API key validity
- Check internet connection
- Monitor API rate limits
- Adjust timeout settings

3. Element Not Found Errors

Symptoms: "Element not found" in browser tests **Solutions**:

- Check website structure changes
- Verify selector strategies
- Increase timeout values

- Use more flexible selectors

4. Memory Issues

Symptoms: Application crashes, slow performance **Solutions**:

- Increase system RAM
- Close unused browser tabs
- Reduce concurrent operations
- Clear session state periodically

Debug Settings

```python
# Enable detailed logging
logging.basicConfig(level=logging.DEBUG)

# Browser debug mode
headless=False  # Show browser window

# Add delays for debugging
await asyncio.sleep(5)
```

API Reference

Core Functions

Website Scraping

```python
scrape_website_sync(url: str) -> dict
```

Parameters:

- `url`: Target website URL (must include https://)

Returns: Structured dictionary with website data

AI Agents

```python
generate_user_stories(site_data: dict) -> str
generate_test_plan(site_data: dict) -> str
generate_test_cases(site_data: dict) -> str
generate_test_data_for_case(testcase_string: str) -> str
```

Browser Automation

```python
async def run_browser_test_agent(
    test_cases_data: list,
    base_url: str,
    scraped_data: dict
) -> tuple[list, list, bool]
```

Returns: (test_results, button_report, error_occurred)

Data Structures

Test Case Structure

```python
{
    'Test Case ID': 'TC-001-01',
    'User Story': 'US-001',
    'Test Scenario': 'Login with valid credentials',
    'Preconditions': 'User is on login page',
    'Test Steps': '1. Enter email\n2. Enter password\n3. Click login',
    'Test Data': 'Email: test@example.com\nPassword: TestPass123',
    'Expected Result': 'User successfully logged in',
    'Actual Result': '',
    'Status': '',
    'Priority': 'High'
}
```

Browser Action Structure

```python
```

```
{
    'action': 'click_element',
    'selector': 'button[type="submit"]',
    'value': 'optional_value',
    'description': 'Click submit button'
}
```

Best Practices

Website Scraping

- Always verify URL accessibility before scraping
- Handle dynamic content with appropriate waits
- Use robust error handling for network issues
- Respect rate limits and server resources

AI Prompt Engineering

- Provide clear, specific instructions to AI agents
- Include examples in prompts for better results
- Validate AI outputs before processing
- Maintain consistent prompt templates

Browser Automation

- Use stable, unique selectors
- Implement proper wait strategies
- Handle dynamic content loading
- Add meaningful error messages
- Take screenshots for debugging

Test Data Management

- Use realistic but safe test data
- Avoid sensitive information in test data
- Maintain data consistency across test cases
- Document test data requirements

Performance Optimization

- Use session state efficiently
- Minimize API calls to LLM
- Implement proper cleanup procedures
- Monitor memory usage during execution

Error Handling

- Implement comprehensive try-catch blocks
- Provide meaningful error messages
- Log errors for debugging
- Graceful degradation when possible

Future Enhancements

Planned Features

1. **Multi-User Testing**: Support for authenticated user workflows
2. **Visual Regression Testing**: Screenshot comparison capabilities
3. **API Testing Integration**: REST API endpoint testing
4. **CI/CD Integration**: GitHub Actions/Jenkins support
5. **Database Testing**: Data validation and integrity checks
6. **Mobile Testing**: Responsive design testing
7. **Accessibility Testing**: WCAG compliance validation
8. **Performance Testing**: Load time and resource monitoring

Technical Improvements

1. **Parallel Execution**: Multiple browser instances
2. **Cloud Browser Support**: Selenium Grid integration
3. **Custom Reporting**: Advanced analytics and insights
4. **Test Scheduling**: Automated execution scheduling
5. **Integration APIs**: External tool connectivity

This documentation provides comprehensive coverage of the AI-Powered Test Automation Suite. For additional support or feature requests, please refer to the project repository or contact the development team.