Automated GUI Testing

Overview

Documents the approach to functional/integration testing for the GUI itself using Selenium.

NOTE: integration testing is only minimally implemented, this mostly presents the methodology and approach that would be used to fully flesh out the automated GUI integration tests.

Coverage

There are General application features, a variety of dashboards, and widgets that can go on the various dashboards. For example,

- General Features
  - Login page
  - Header
  - Footer
- Dashboards
  - Main Dashboard
  - Project Details Dashboard
  - Map Records Dashboard
  - Mapped Concept Dashboard
  - Feedback Conversation Dashboard
  - Index Viewer Dashboard
  - Edit Map Record Dashboard
  - etc.
- Widgets
  - Map Projects Widget
  - Available Work Widget
  - Assigned work Widget
  - Feedback Widget
  - etc.

Complete coverage would address each mode of use of general features and each widget that appears on each dashboard.

Coverage Definition

The attached spreadsheet formally defines automated GUI QA coverage from an integration testing perspective.

Assumptions

- A dev database exists that is loaded with exactly the development content from the mapping-service-data git project.
- Any changes made are reverted to initial conditions so all tests can assume identical initial conditions

Coverage is defined in a multi-dimensional matrix that takes into account the services, the methods of the services, and the modes of use of the services (including normal use, degerate use, and edge cases). Coverage also considers combinations of calls that have related effects to each other and a placeholder for "ad hoc" tests that can be filled in as gaps in coverage are identified.

The following sample coverage spreadsheet begins to outline the methodology for defining coverage though it is not fully fleshed out.

- Automated GUI Coverage Spreadsheet

Organization

Integration tests are organized under the integration-testing project. Looking through the Java packages, you should find a "org.ihtsdo.otf.mapping.test.selenium" package that defines a number of test classes that represent the coverage spreadsheet.

Each column of each tab of the coverage spreadsheet is represented by a single testing class, where each row within that column corresponds to a particular method of that testing class. The coverage spreadsheet defines both the class names used to implement tests as well as the method names (so they can be easily cross referenced).

The test names themselves are "test cases" that are organized into "test suites" according to the structure of the spreadsheet.

For each test suite and test case, there should also exist documentation. A test suite is a collection of test cases and a test case is a "script" detailing the actions involved and the expected outcomes with any other needed information.

Integration Test Suites

Test suites are organized by spreadsheet tab.

- General GUI Test Suite
- Map Project Widget Test Suite

- Available Work Widget Test Suite
- ... other GUI test suites ...
- Combo Jpa Test Suites
    - TBD
- Ad Hoc Jpa Test Suites
    - TBD

Implementation Using Selenium

Automated GUI testing is accomplished via Selenium in the integration-tests package. Selenium enables automated interaction with HTML DOM objects, including form completion and submission, button clicking, and element search.

To properly interact with the application, elements should be assigned unique identifiers in the application. This is not comprehensively done across the application yet. As test cases are developed, identifiers should be assigned to elements that need them to complete the test. While selenium supports identifying elements by means other than id, it is strongly recommended that only the ID mechanism be used. It is more stable across time and continued development of the application.

Requirements

The following conditions are required for Selenium automated GUI testing to function

- The server must be running
- The Mapping Tool's database must exist
- The valid user must exist in the database. If authentication is enabled, the user must be able to be authenticated
- The configuration parameters must be set in the configuration properties file.

Configuration Parameters

The integration test is run with a -Drun.config parameter pointing to a config file that contains the following configuration properties.

- selenium.browser: The desired browser. Accepted values are:
    - firefox – Mozilla Firefox
    - chrome – NOTE that Chrome is not automatically supported by Selenium. If testing in Chrome is desired, see the Selenium homepage for instructions.
    - ie – Internet Explorer
- selenium.timeout: The length of time (in ms) that the tests will attempt to validate DOM/Javascript elements
- selenium.user.guest.name: The username (i.e. "guest") for the Mapping Tool's guest user
- selenium.user.valid.name: The username of a valid user. **This must exist. If security is enabled, authentication must succeed for this user.**
- selenium.user.valid.password: The password of the valid user.
- selenium.user.invalid.name: A fake username. This must not exist in the database. A fake password is auto-generated.

For example:

Sample Configuration

```
#
# Selenium UI testing parameters
#
selenium.browser = firefox
selenium.timeout = 10000
selenium.user.guest.name = guest
selenium.user.valid.name = EDIT_THIS
selenium.user.valid.password = EDIT_THIS
selenium.user.invalid.name = not_a_real_user
```

Selenium Interaction with AngularJS

AngularJS and Selenium are not natively compatible. Specifically, Selenium does not by default wait for dynamic DOM manipulation by Angular.

Two sections of code are displayed in the code block below:

- In the first (functional) section, the WebDriver will successfully find the element by its id, but then waits for text to be dynamically inserted. The length() check ensures that the integration test does not proceed until text is actually substituted into the DOM element specified.
- In the second (non-functional) code section, the WebDriver will successfully find the element by its id, but will return an empty string immediately, as Angular has not yet injected the requested text.

Examples of Handling Selenium and AngularJS Injection Timing

```
// THIS WORKS
(new WebDriverWait(webDriver, new Long(
        config.getProperty("selenium.timeout")) / 1000))
        .until(new ExpectedCondition<Boolean>() {
          public Boolean apply(WebDriver d) {
            return webDriver.findElement(By.id("[ELEMENT NAME]")).getText()
                .length() > 0;
          }
        });

// THIS DOES NOT WORK
(new WebDriverWait(webDriver, new Long(
        config.getProperty("selenium.timeout")) / 1000))
        .until(new ExpectedCondition<Boolean>() {
          public Boolean apply(WebDriver d) {
            return webDriver.findElement(By.id("[ELEMENT NAME]")).getText();
          }
        });
```

References/Links

- Selenium - seleniumhq.org
- http://en.wikipedia.org/wiki/Selenium_(software)