

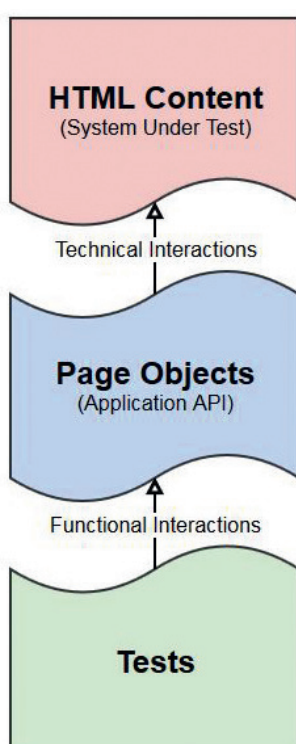
WebTester 2.0 – Your Open Source Web Application Automation Framework

WebTester is an open source automation framework for web applications. It is based on Selenium (<http://www.seleniumhq.org>). Years of working with several different automation tools have shown us the weaknesses most of these frameworks suffer from. Overly generic APIs, missing extension points or simply strange design choices. We decided to provide an intuitive, declarative and extensible API for writing effective and maintainable tests in Java. All of this is built on top of Selenium – in our opinion the number one test driver.

Top Features

- Optimized for Java 8 and above
- Intuitive Page Object Pattern with simple annotation-driven element identification
- Useful predefined element classes (e.g. Button, TextField, ...)
- Simple API for runtime element identification
- Boost reuse with easy composition of pages and page fragments
- Highlighting of used elements for visual debugging
- Custom event handling: from a simple screenshot on exception to custom report generation
- Seamless integration with common frameworks like: AssertJ, Hamcrest, JUnit, Spring, etc.
- Selenium is always just a method call away!


Page Object Pattern



WebTester is designed around the Page Object pattern. You model your web application's pages as objects and possible user interactions as methods. These page objects provide a stable layer between your tests and volatile HTML pages, making your tests much easier to maintain. Whenever you change the content on a web page, you simply update the corresponding page object, and all your tests will continue to work.

Example Application

Our application uses a simple login screen. The user should be able to login with valid credentials. If either user name or password are incorrect, an error should be displayed.



The example application shows two states of a login screen. In the first state, the "Username:" field contains "tester_de" and the "Password:" field contains masked characters. A "LOGIN" button is visible. An arrow points to the second state, which shows a "Welcome!" message and "You successfully logged in!". A second arrow points to a third state, which shows an "Error" message: "Unknown user: 'unknown'". The login form is shown again with "unknown" in the username field and masked password, with a "LOGIN" button.

Declaring Pages

To test the login screen, we first declare three page objects in WebTester: Login, Welcome and Error.

Here's what's happening in detail:

1. Pages are interfaces which extend the „Page“ base interface. Why interfaces and not classes? Java interfaces support multiple inheritance, which we can use to easily compose larger pages.
2. „@IdentifyUsing“ declares a page fragment (here, the TextField for the user's name). Per default we use CSS Selectors to identify elements, but other methods are supported as well: XPath, ID, Tag Name, Link Text, etc.
3. This method returns the previously declared „TextField“ for the user's name. The implementation is automatically supplied by WebTester.
4. Methods annotated with „@PostConstruct“ run before any tests and can validate preconditions, e.g. that the page is actually displayed in the browser.
5. WebTester supports all popular assertion frameworks. Here we use AssertJ to assert that the field for username is visible.

Login Page

```
1 public interface LoginPage extends Page { // (1)
2
3     @IdentifyUsing("#username") // (2)
4     TextField usernameField(); // (3)
5     @IdentifyUsing("#password")
6     PasswordField passwordField();
7     @IdentifyUsing("#login")
8     Button loginButton();
9
10    @PostConstruct // (4)
11    default void correctPageIsDisplayed() {
12        assertThat(usernameField()).isVisible(); // (5)
13        assertThat(passwordField()).isVisible();
14        assertThat(loginButton()).isVisible();
15    }
16
17    default WelcomePage login(Credentials credentials) { // (6)
18        return setUsername(credentials.getUsername())
19            .setPassword(credentials.getPassword())
20            .clickLogin();
21    }
22
23    default ErrorPage loginExpectingError(Credentials credentials) {
24        return setUsername(credentials.getUsername())
25            .setPassword(credentials.getPassword())
26            .clickLoginExpectingError();
27    }
28
29    default LoginPage setUsername(String username) { // (7)
30        usernameField().setText(username);
31        return this;
32    }
33
34    default LoginPage setPassword(String password) {
35        passwordField().setText(password);
36        return this;
37    }
38
39    default WelcomePage clickLogin() { // (8)
40        loginButton().click();
41        return create(WelcomePage.class);
42    }
43
44    default ErrorPage clickLoginExpectingError() { // (9)
45        loginButton().click();
46        return create(ErrorPage.class);
47    }
48
49 }
```

6. This is a workflow method. It uses fluent style to describe a chain of user interactions with the page (valid login in this case). Workflow methods are composed of simpler methods that describe single interactions with a page. Like all methods that describe user interactions, it returns a page object instance (in this case, the welcome page displayed after successful login).
7. This is a state changing method. It describes entering the user's name into the username text field. State changing methods always return the updated page object of the current page.
8. This is a navigation method. It describes clicking on the login button and expecting to be routed to the welcome page. Navigation methods generally describe the user's expectation of the next page being displayed, which is why they return a different page object.
9. Applications can display different pages depending on the context (e.g. valid vs. invalid credentials used at login). Navigation methods describe which page is expected next,

so there can be multiple navigation methods for the same basic interaction (clicking the login button), each with a different expected outcome.

Welcome Page

```
1 public interface WelcomePage extends Page {
2
3     @IdentifyUsing("#welcomeMessage")
4     @PostConstructMustBe(Visible.class) // (1)
5     GenericElement welcomeMessage(); // (2)
6
7     default String getWelcomeMessage() { // (3)
8         return welcomeMessage().getVisibleText();
9     }
10
11 }
```

The Welcome page object demonstrates some more features:

1. The „@PostConstructMustBe“ annotation can be used as an alternative for „@PostConstruct“ to ensure that the test's preconditions are met. In this case, the welcome message element has to be visible.
2. In case you don't care about the specific functional representation of an element, you can always use the „GenericElement“ type to access it. You can freely interact with a GenericElement, but there won't be sanity checks (e.g. you can send key presses to a button, but also to a link).
3. This is a getter method. These methods usually return information from the displayed page. In this case it returns the text of the welcome message.

Error Page

```
1 public interface ErrorPage extends Page {
2
3     @IdentifyUsing("#error")
4     @WaitUntil(Visible.class) // (1)
5     Error error(); // (2)
6
7     default String getErrorMessage() {
8         return error().getMessage();
9     }
10
11 }
```

The Error page object shows how WebTester handles asynchronous page updates:

1. Elements can be annotated with „@WaitUntil“ and a condition. Each time the element is accessed, the test will wait until the specified condition is met. Here, the error message is not returned until after the element becomes visible. Of course, things can go wrong and the element may never be displayed. In such a case, WebTester waits for a certain configurable period, before the element access fails due to a timeout.
2. The page fragment described here is called „Error“, which is not one of WebTester's standard fragments. We explain how to define custom fragments in the next section.

Defining Your Own Page Fragments

Elements on a web page are called „page fragments“ in WebTester. You can easily define your own custom page fragments. They can either represent a single element of your application (e.g. a button) or a group of elements which build a logical context (e.g. a text field and a button, giving you a search widget). This nesting of page fragments maps nicely to the nesting of HTML elements.

Custom „Error“ Fragment

```
1 @Mapping(tag = "div", attribute = "class", values = "error") // (1)
2 public interface Error extends PageFragment { // (2)
3
4     default String getMessage() {
5         return find("#message").getVisibleText(); // (3)
6     }
7
8 }
```

This is the definition of the Error page fragment:

1. Page fragments are mapped to HTML code. In this case we define that an „Error“ has to be a <div> with a class attribute of „error“. In case the „Error“ fragment is ever used for another HTML tag, an exception will be thrown.
2. Page fragments extend the base interface „PageFragment“.
3. Instead of explicitly defining a nested page fragment, you can look them up dynamically. Here we use our „ad-hoc finding“ API to look up an element within the context of the Error page fragment.

Writing Tests

The actual test for the Login page is written in JUnit 5 and uses features provided by WebTester's extensions.

The test methods themselves are pretty simple. There is however some WebTester-specific setup to do:

1. With this convenience annotation all of WebTester's JUnit 5 extensions are activated for the current test class.
2. This extension handles the startup and shutdown of our web application. It is not a part of WebTester. We generally recommend that your tests handle the lifecycle of the application instance under test.
3. @Managed tells WebTester to automatically handle initialization and shutdown of the annotated browser. It is provided by the lifecycle management extension of WebTester.
4. @CreateUsing configures a factory class that WebTester will use to create browser instances. The kinds of browser used for testing and details of Selenium's web driver configuration are managed by these factories.

Login UI-Test

```
1 @EnableWebTesterExtensions // (1)
2 @ExtendWith(EmbeddedApplication.class) // (2)
3 public class LoginTest {
4
5     @Managed // (3)
6     @CreateUsing(FirefoxFactory.class) // (4)
7     @EntryPoint("http://localhost:8080/login") // (5)
8     static Browser browser; // (6)
9
10    @Initialized // (7)
11    LoginPage loginPage; // (8)
12
13    @Test
14    void loginWithExistingCredentials() {
15        Credentials credentials = Credentials.builder()
16            .username("tester_de")
17            .password("123456")
18            .build();
19        WelcomePage welcomePage = loginPage.login(credentials);
20        assertThat(welcomePage.getWelcomeMessage()).isEqualTo("You successfully logged in!");
21    }
22
23    @Test
24    void loginWithUnknownUser() {
25        Credentials credentials = Credentials.builder()
26            .username("unknown")
27            .password("123456")
28            .build();
29        ErrorPage errorPage = loginPage.loginExpectingError(credentials);
30        assertThat(errorPage.getErrorMessage()).isEqualTo("Unknown user: 'unknown'");
31    }
32
33    @Test
34    void loginWithWrongPassword() {
35        Credentials credentials = Credentials.builder()
36            .username("tester_de")
37            .password("wrong")
38            .build();
39        ErrorPage errorPage = loginPage.loginExpectingError(credentials);
40        assertThat(errorPage.getErrorMessage()).isEqualTo("Wrong password!");
41    }
42
43 }
```

5. @EntryPoint provides a URL for the browser to open before each test. This URL does not have to be static, it can include placeholders for configuration properties!
6. A „Browser“ is an abstraction which provides methods for interacting with any supported browser (like Firefox, Chrome, Internet Explorer etc.).
7. @Initialized tells WebTester to initialize the page object before each test.
8. The initialized page object is our previously defined „LoginPage“. The page object matches the page displayed in the browser, which we specified as entry point.

What's Next?

What we showed you is only a subset of the features provided by WebTester. For a full overview take a look at our documentation on GitHub. For additional details of how to set up WebTester for your project, check out the demo application here: <http://www.novatec-gmbh.de/wt-demo>

Contact



Anis Ben Hamidene

Senior Managing Consultant

Head Of Competence Area Agile Quality Engineering

Phone: +49 711 22040-700

Mobile: +49 170 7998 715

Mail: anis.benhamidene@novatec-gmbh.de

WebTester is created and developed by NOVATEC

Founded in 1996, NovaTec Consulting GmbH is an independently managed IT consulting firm based in Germany. The service portfolio ranges from individual needs analyses, architecture consulting, development of high quality software systems, as well as the selection and implementation of high performance software applications. As experts in agile development methods we produce high quality products as well as helping our clients introduce and improve their own agile development methods.

NovaTec Consulting GmbH
Dieselstraße 18/1
D-70771 Leinfelden-Echterdingen
www.novatec-gmbh.de



Twitter

@NT_AQE



Website

www.novatec-gmbh.de/wt



Blog

www.novatec-gmbh.de/aqe-blog



YouTube

www.novatec-gmbh.de/wt-youtube



GitHub

www.novatec-gmbh.de/wt-github



Trainings

Agile Testing: www.novatec-gmbh.de/at-training

Developer Testing: www.novatec-gmbh.de/dt-training