Quick Starting Guide to Ipreo Automation Framework

# Preparation steps

## Environment configuration

1. Verify that you have Visual Studio 2012 or higher installed. (Note: all menu items in this guide are named as in VS 2013).
2. **(For EPAM employees only! Ignore that if you work for Ipreo directly and/or use the Ipreo network)** Add the following line to hosts file (C:\Windows\System32\drivers\etc):

172.31.6.66 mpgtc30ny1us01

## Visual Studio configuration

1. Add new package source to download AutomationFramework from it.
   1. Go to Tools -> Library Package Manager -> Package Manager Settings
   2. Select Package Manager -> Package Sources
   3. Click ‘+’ button and modify the new entry:

Name: on your choice, suggested “Ipreo”.

Source: <http://mpgtc30ny1us01:8000>

1. (Optional): if you need SpecFlow for your tests, see [SpecFlow](#_SpecFlow) section for additional instructions (SpecFlow requires not only a NuGet package, but also an extension for Test Manager).

## Solution and project configuration

1. Create a new project (suggested type: Class Library; use Unit Test Project if you are going to use Microsoft Unit Testing framework)
2. Install NuGet packages with AutomationFramework for the project:
   1. Open Solution Explorer window (View -> Solution Explorer)
   2. In Solution Explorer, right-click on the project -> Manage NuGet Packages…
   3. Select ‘Online’ -> ‘Ipreo’ and add the following projects:
      1. **Ipreo.AutomationFramework.UI** – contains tools to manage browser, pages, page elements etc.
      2. **Ipreo.AutomationFramework.UI.Components** – contains predefined element types (HtmlButton, HtmlTextField etc.)
      3. (Optional) **Ipreo.AutomationFramework.UI.Selenium** if your project is Selenium-based.
3. Install NuGet packages for additional dependencies:
   1. For NUnit, see [NUnit](#_NUnit) section.
   2. For SpecFlow, see [SpecFlow](#_SpecFlow) section.
   3. For MsTest, see [Microsoft Unit Testing Project](#_Microsoft_Unit_Test) section.

### SpecFlow

1. Install NuGet packages -> install **SpecFlow.**
2. Select main menu item Tools -> Extensions and Updates -> Online -> Install **SpecFlow for Visual Studio \*\*\*\*** extension.

### NUnit

1. For NUnit tests, install NuGet package **NUnit Framework** from nuget.org.
2. Select main menu item Tools -> Extensions and Updates -> Online -> Install **NUnit3 Test Adapter** extension.

### Microsoft Unit Test Project

1. For SpecFlow integration, install NuGet package **Specflow.MsTest**.

## Suggested project structure

The suggested folder structure for the project is provided below. Feel free to adjust it to your needs.

1. **Templates** – a folder for base classes such as BaseTest (with default setup/teardown actions), BasePage, BaseContext etc.
2. **Features / Tests** – a folder to store feature files (for BDD tests) or test classes (for DSL tests).
3. **Steps (StepDefinitions)** – a folder to store step definitions for BDD tests.
4. **Contexts** – a folder to store classes which describe certain flow, like Login flow, Search flow etc. For more details see [Context](#_Context) section.
5. **Pages** – a folder to store page classes.
6. **Extensions** – a folder to store classes which extend the framework capabilities and might be incorporated into it in future (e.g. new element definitions, helper libraries etc.). You can find out more information about new elements in [Elements and Behaviors](#_Elements_and_behaviors) section.

## App.config file

Framework can be configured using App.config file.

# Tests Development

## Pages and Complex Controls / Containers

### Overview

**Page** is a high-level container for all elements displayed on the screen, excluding those we are not interested in. Page can contain standalone elements and containers.

**Container**, or Complex Control, is a class, which describes some group of elements on the page. It can contain elements and the other containers.

Containers are very useful for:

* Repeatable element groups
* Element groups located under common parent element

### Page and Container Example

The page on screenshot below contains multiple settings widgets. We can declare a SelectorPanel container class to use it multiple times on the page.

public class SelectorPanel : Container

{

[FindBy(How.ClassName, "headerCell")]

public HtmlLabel Header;

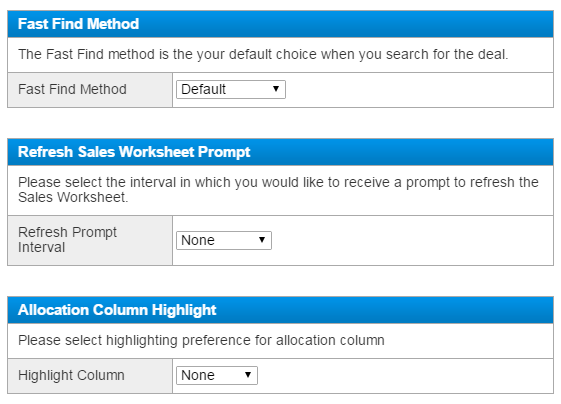
[FindBy(How.ClassName, "resultsRowB")]

public HtmlLabel Description;

[FindBy(How.TagName, "select")]

public HtmlDropDown Values;

}



The corresponding Page class:

public class SettingsPage : Page

{

[FindBy(How.Xpath, "<…>")]

public SelectorPanel FastFindMethod;

[FindBy(How.Xpath, "<…>")]

public SelectorPanel RefreshSalesWorksheetPrompt;

[FindBy(How.Xpath, "<…>")]

public SelectorPanel AllocationColumnHighlight;

}

### How to create a Page class

1. Create a class, which extends Page class from AutomationFramework.

public class LoginPage : Page

1. Add some elements to your page.

public HtmlTextField LoginField;

public HtmlTextField PasswordField;

public HtmlButton LoginButton;

1. Add locator attributes to determine unique element position.

[FindBy(How.Id, "loginInput")]

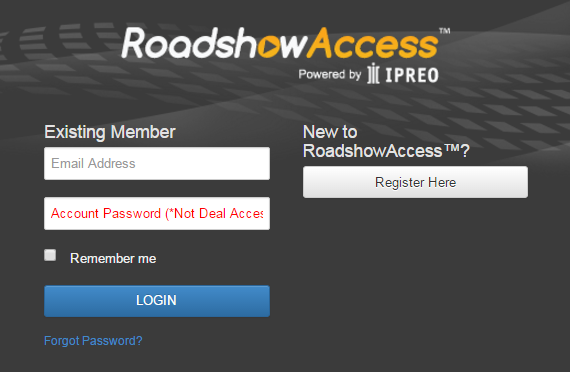
public HtmlTextField LoginField;

[FindBy(How.Id, "Password")]

public HtmlTextField PasswordField;

[FindBy(How.CssSelector, ".btn-sm.btn-primary")]

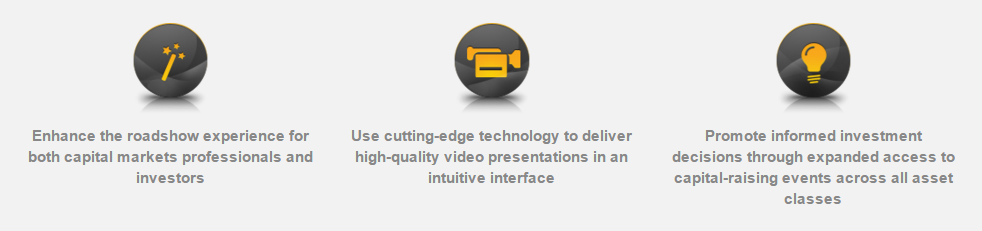
public HtmlButton LoginButton;



1. **Advanced**: if you cannot find a unique locator for your element (or it looks bad, like a long XPath), add filters to make your search more specific. Note: you can use any number of filters, even of the same type.
   1. [IsDisplayed] or [IsDisplayed(false)] – element should be visible/invisible
   2. [HasAttribute(name, value)] – element has attribute **{name}** with value **{value}. Note:** you can use multiple different HasAttribute filters.
   3. [WithTimeout("00:00:10")] – element requires specific timeout, different from default one
   4. [NoCaching] – element will never be used twice, it will always be looked up from scratch for each interaction.
2. Add collections of elements (you will be able to iterate through it and access specific elements by index).

[FindBy(How.ClassName, "col-sm-4")]

public IElementCollection<HtmlLink> MarketingIcons;



1. Add containers (see [Containers](#_How_to_declare) section).

public DealListArea DealsList;

1. Add locator attributes to container field if all its elements are located under the same parent element.

[FindBy(How.Id, "dealListArea")]

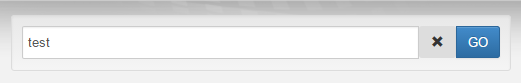
public DealListArea DealsList;

1. Proceed to [How to use your new Page or Container](#_How_to_use).

### How to create a Container class

1. Create a class, which extends Container class from AutomationFramework.

public class DealSearchBlock : Container



1. Add elements / collections of elements / other containers, as described in [Page](#_How_to_declare_1) section.

[FindBy(How.TagName, "input")]

[IsDisplayed]

public HtmlTextField SearchField;

[FindBy(How.TagName, "button")]

[IsDisplayed]

public HtmlButton GoButton;

[FindBy(How.CssSelector, ".clr-btn")]

public HtmlButton ClearButton;

1. (Optional) Add simple methods involving only container elements – no other containers or pages should be used.

public void FindDeal(string dealName)

{

SearchField.SetText(dealName);

GoButton.Click();

}

1. Place Container onto the Page with a proper parent element locator (if needed). For more details, see [How to use your new Container as a Page field](#_How_to_use_2) and [How to use your new Container as a standalone object](#_How_to_use_3) sections.

public class DashboardPage : Page

[FindBy(How.ClassName, "input-group")]

[IsDisplayed]

public DealSearchBlock DealSearch;

[FindBy(How.Id, "dealListArea")]

public DealListArea DealsList;

}

### How to use your new Page

1. Declare the Page in your Test or Context class.

public class LoginContext

{

private LoginPage loginPage;

1. Create Page instance in context constructor or test configuration method:

public LoginContext()

{

loginPage = PageService.Create<LoginPage>();

}

**Note 1:** it is enough to create a page once before all tests. It does not have to be present on the screen for that!

**Note 2:** do **not** create Page instance in line with declaration, e.g. private LoginPage loginPage = PageService.Create<LoginPage>(); – it will fail, because PageService will not be ready by that moment.

1. Access all public fields, properties and methods of the page as with a regular class.

### How to use your new Container as a Page field

1. Add your Container to the Page along with Elements.

public class LoginForm : Container{…}

public class LoginPage : Page

{

public LoginForm LoginForm;

}

1. If your Container has a parent element, specify it using FindBy and FilterBy parameters, just like you do with regular elements:

public class LoginPage : Page

{

[FindBy(How.Id, "loginForm")]

public LoginForm LoginForm;

}

In this case, all fields of LoginForm will be looked up starting from the element with ID=”LoginForm”, not the page root.

### How to use your new Container as a standalone object

## Elements and Behaviors

### Overview

There is a list of predefined classes, such as HtmlTextField, HtmlLabel etc. You can also create new element types in static way (by creating new classes) or dynamic way (by applying new behaviors). Behaviors consist of Traits.

**Trait** is some user action allowed for the element.

**Example**:

* ITextReadable – user can read text from the element (label / link / text box /etc.)
* ITextEditable – user can write text (drop-down / text box / etc.)
* ISelectable - user can select some value from the list

Traits can be used in multiple behaviors, because many behaviors involve similar user actions.

Traits are interfaces with a set of allowed methods.

**Example:**

public interface IHasPlaceholder

{

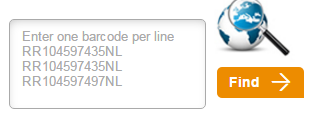
string GetPlaceholder();

bool WaitForPlaceholder(string expectedValue);

}

**Behavior** is what user can do with the element. It describes how element reacts to user actions available in some context. It can consist of several actions (traits).

Example:



An input field on the screenshot demonstrates 2 behaviors:

* **ITextFieldBehavior**, because user can type some text into it, clear it, etc.
* **IPlaceholderBehavior,** because this field has a placeholder, which can be read.

**ITextFieldBehavior** consists of 2 traits: ITextEdtable (can write) and ITextReadable (can read).

**IPlaceholderBehavior** consists of 1 trait: IHasPlaceholder.

### How to create a Behavior

1. (Optional) Create a trait.

If there is some new user action behind the behavior (e.g., user double-clicks on some objects), you can create a trait to use it in different behaviors.

A trait is simply an interface with a set of methods specific for its user action.

public interface ICanHaveFocus

{

void Focus();

void Unfocus();

bool IsFocused();

}

1. Create a behavior interface.

We need interfaces, because user actions are implemented in different way for different automation tools (Selenium, CodedUI, etc.), but behaviors are common to all elements regardless to the tool.

To define a behavior interface:

* Extend some trait(s)
* Extend basic interface **IElementBehavior**

public interface IFocusBehavior : ICanHaveFocus, IElementBehavior

{

}

1. Create a behavior implementation.

To implement your behavior:

* Extend Base\*\*\*ElementBehavior, where \*\*\* = Selenium | CodedUI | etc.
* If your behavior does not depend on the tool (as in case of Tables), use BaseToolIndependentBehavior
* Extend your behavior interface

public class SeleniumFocusBehavior : BaseSeleniumElementBehavior, IFocusBehavior

{

For tool-dependent behaviors, you will get access to properties:

* **ElementDecorator** – Framework-specific base element instance
* **SeleniumElement**– Tool-specific element instance (i.e. IWebElement for Selenium)

You can use these properties to gain access to element attributes and methods.

For tool-independent behavior, you will get access to **Element** property, which is a tool-independent element (i.e. same for CodedUI, Selenium etc.).

public void Focus()

{

const string focusCommand = "arguments[0].focus();";

DriverService.Driver.ExecuteScript(focusCommand, NativeElement);

}

1. Apply your new behavior to any element (see [How to use a Behavior](#_How_to_use_1) section).

### How to use a Behavior

You can add any behaviors dynamically to any element. For example, if you have a test, which checks placeholder in some field, you can obtain this placeholder in this way:

HtmlTextField SearchField;

<…>

string placeholder = SearchField.AddBehavior<IPlaceholderBehavior>().GetPlaceholder();

**Note that this behavior won’t be saved in the element!** To keep its reference, you have to declare a separate variable:

IPlaceholderBehavior behavior = SearchField.AddBehavior<IPlaceholderBehavior>();

or

IPlaceholderBehavior behavior = SearchField.WithBehavior<IPlaceholderBehavior>();

You can apply any number of behaviors you need, one by one.

public void TypeIntoSearchField(string value)

{

//Type text by sending keystrokes for each symbol in {value}

SearchField.WithBehavior<ISimulateUserInputBehavior>().SetText(value);

//Use JS to remove focus from the element

SearchField.WithBehavior<IFocusBehavior>().Unfocus();

}

public void PasteIntoSearchField(string value)

{

//Paste text from clipboard by pressing Ctrl-V

SearchField.WithBehavior<ICopyPasteBehavior>().SetText(value);

//Use JS to remove focus from the element

SearchField.WithBehavior<IFocusBehavior>().Unfocus();

}

In general, if you need to perform any specific non-default action on some element, you can use the following syntax:

SomeField.AddBehavior<ISomeBehavior>().PerformActionFromSomeBehaviorScope();

or

SomeField.WithBehavior<ISomeBehavior>().PerformActionFromSomeBehaviorScope();

### How to create a new Element type

If you want to use some combination of behaviors often, you can create a new element type.

Example: a field which always contains some placeholder when empty.



1. Create a class, which extends ElementBase or some predefined element classes (e.g. HtmlLabel, HtmlButton etc.). Make it implement all behavior interfaces you need.

public class HtmlFieldWithPlaceholder : Element, ITextFieldBehavior, IPlaceholderBehavior

(Let’s pretend we don’t yet have HtmlTextField class and extend Element instead, in order to make the example more general!)

1. Add private properties for all behaviors:

private ITextFieldBehavior \_textFieldBehavior;

private IPlaceholderBehavior \_placeholderBehavior;

1. Setup them in class constructor:

public HtmlFieldWithPlaceholder()

{

\_textFieldBehavior = AddBehavior<ITextFieldBehavior>();

\_placeholderBehavior = AddBehavior<IPlaceholderBehavior>();

}

1. Redirect all corresponding Element methods to behavior methods with the same names:

public string GetText()

{

return \_textFieldBehavior.GetText();

}

public string GetPlaceholder()

{

return \_placeholderBehavior.GetPlaceholder();

}

…and also SetText(), AppendText, Clear() and WaitForPlaceholder().

1. You are done! Use your new field in page classes. All its methods are accessible and ready for use.

public class SmartSearchPage

{

[FindBy(How.Id, "smart-search")]

public HtmlFieldWithPlaceholder SmartSearchField;

}

1. **Note:** try to avoid creating too many element classes like:
   * HtmlTextFieldWithPlaceholder
   * HtmlTextFieldWithPlaceholderAndIcon
   * HtmlTextFieldWithButton
   * HtmlTextFieldWithUnicorn

If you need to use it in 1-2 places, consider using simple element type and adding behavior over it instead of creating a whole new class.

### How to find an Element

We can not only use Elements which are Page fields, but also initialize them dynamically in tests.

To find one element:

HtmlTextField loginField = new HtmlTextField().FindBy(How.TagName, "input");

…or:

HtmlTextField loginField = ElementService.Find<HtmlTextField>().FindBy(How.TagName, "input");

To find multiple elements:

IElementsCollection<HtmlButton> serviceButtons = ElementService.FindAll<HtmlButton>()

.FindBy(How.ClassName, "service-button");

To find element with filters (constraints):

IElementsCollection<HtmlButton> serviceButtons = ElementService

.FindAll<HtmlButton>()

.FindBy(How.ClassName, "service-button")

.FilterBy(new IsDisplayed())

.FilterBy(new HasAttribute("someAttribute", "someValue"));

To find element with timeout:

IElementsCollection<HtmlButton> serviceButtons = ElementService

.FindAll<HtmlButton>()

.FindBy(How.ClassName, "service-button")

.FilterBy(new WithTimeout(TimeSpan.FromSeconds(10)));

## Contexts

### Overview

**Context** is a class, which contains mini-scenarios related to certain flow. The pages, which are involved in the flow, are the fields of this class. The scenarios are its public methods.

Example: LoginContext can contain multiple methods related to login procedure:

public class LoginContext

{

private LoginPage loginPage;

public LoginContext()

{

loginPage = PageService.Create<LoginPage>();

}

public LoginPage OpenApplication(){…}

public void LoginAs(string username, string password){…}

}

### Q&A

**Why do we need context? We can use Test/Step Definition or Page class to store these methods.**

It is a good solution for two reasons:

* Page should not know about any other pages except itself! Otherwise, if page A knows about B, and B is changed, we will have to change A too. If scenarios involve more than 1 page or container, it is a bad idea to place them to Page classes.
* Any context can be used in multiple tests, and if you place these methods with scenarios into Test/Step Definition classes, other Tests/Step Definitions will not be able to use them.

**Is it safe to create a page in Context constructor, like in example above? What happens if I close or refresh the page?**

Don’t worry. Our Page is a smart “lazy” model, which only communicates with real browser, when you try to do something with its elements (e.g., check their visibility).It will be enough if the page related to Page class is present on the screen when you use something from Page class.

**Can I write the following in Context?**

public class LoginContext

{

private LoginPage loginPage = PageService.Create<LoginPage>();

Unfortunately, no. PageService will not be ready at that moment.

## Tests (no BDD)

Below you will find a short list of best practices for DSL-tests (i.e. not BDD).

1. A good test is a test, which can be read and understood by a manual tester. (All words form a readable text in English)

**Good style:**

sharedSteps.LoginAsAdministrator();

**Bad style:**

new HtmlTextField().FindBy(How.Id, "field1").SetText("QAuser");

1. A good test is a short test.

How do we check that our test is short?

We can consider 3 test step types:

* Navigation (N) – preparation steps, we go to some application area. Example: open user profile page
* Action (A) – we perform the action, which this test is testing. Example: update user nickname
* Verification (V) – we verify the Action result. Example: verify that user nickname is updated.

**Perfect test formula**: N – A – V.

**Normal test formula**: NNN – AAA – VVV or NA – NA – NA – VVV.

If you test contains more steps, it will be difficult to support it.

1. A good test belongs to some active test suite.

If you never execute this test, it is a useless test.

Group your tests into test suites and use test categories like @Smoke, @Long, @Integration etc. For more details, see [Test Categories](#_Test_Categories) section.

1. A good test contains no **for**’s, **if**’s, **while**’s, **try**’s etc.

**Why no if’s?**

Because it means we will have a fork in our test case, while a test should go one way. Create two tests for different execution paths.

**Why no loops?**

A loop indicates that in order to do some action from business or testing perspective, we have to repeat certain action N times. A test should consist of business-related steps. Example: let’s say user account locks after we’ve entered invalid password for 5 times. The test will be much more readable if we create a method like LockUserAccountByEnteringInvalidPasswords(). then if we write a loop. Besides, if the requirements change, it will be easier to fix the flow.

**Why no try/catch?**

These statements influence execution order (some lines in try block might be skipped, which creates an unwanted fork), and they generally suppress exceptions, leaving you with much less information for investigation in case of failure.

1. A good test always contains assertions. No assertion means test checks nothing.
2. A good test does not duplicate another test functionality, especially assertion. If we check that application search works in one test suite, there is no need to verify that additionally in another test suite.

## Tests (BDD)

### Features

### Step Definitions

## Test Categories

A test has no value if it isn’t executed regularly. In order to support regular test execution, it is good to use specific test categories (using NUnit or SpecFlow) to organize tests into groups. The same test can have one or multiple categories assigned.

### Smoke tests

These tests will be executed each time the new build is rolled out. They should be short and not end-to-end, because end-to-end tests have poor stability.

### Area tests

These tests are related to specific application area, which is useful during regression testing.

### Component/integration tests

These tests verify certain parts of the framework, like template pages, new behaviors, custom utilities etc. This test suite has to be executed locally before each check-in operation.

## Utilities

### PageService: Create a Page or Container instance

PageService creates instances of Pages and Containers.

To create a page:

LoginPage page = PageService.Create<LoginPage>();

To create a container without parent element:

LoginForm form = PageService.Create<LoginForm>();

To create a container with parent element:

Element formContainer = new Element().FindBy(How.Id, "login-form");

LoginForm form = PageService.Create<LoginForm>(formContainer);

### ElementService: Find element(s)

See [How to find an Element](#_How_to_find) section.

### SettingsService: Manage timeouts

Timeout for Page to be loaded:

SettingsService.TimeoutSettings.PageTimeout = …;

Timeout for Element to be found:

SettingsService.TimeoutSettings.ElementTimeout =…;

Timeout for operation to be retried (if we failed to do something at first, but in general have some more time to do that – e.g. find an Element or wait for it to gain some state):

SettingsService.TimeoutSettings.RetryTimeout =…

### SmartWait: Wait until something happens… or not

In general, SmartWait can be used in the following situation:

* You want to check some condition for a test object (Page, Container, Element etc.)
* You know this condition won’t be satisfied at once (for example, element will be absent on the page at first, but it will appear later)
* If the condition is not satisfied within some period (say, 1 minute), you want to stop checking and return “false”
* If the condition is satisfied, checking process stops immediately and returns “true”

Imagine we have 2 elements of different types:

SomeElementType element1 = new SomeElementType();

AnotherElementType element2 = new AnotherElementType();

We should only continue our test if the application meets 2 conditions: element1 has value “Ready” in SomeField, and element2 has value 1 in AnotherField.

First, we will create SmartWait instance:

SmartWait wait = new SmartWait();

Then we will describe our conditions:

wait.WaitUntil<SomeElementType>(element1, element => element.SomeField == "Ready");

* WaitUntil<SomeElementType> means that we will check condition for SomeElementType
* element1 is the instance of SomeElementType that we want to check
* element => means we declare a variable element of type SomeElementType, and element1 will be its value
* element.SomeField == "Ready" is a condition we want to check – it is easy to notice that it should return “true” or “false”, equals or not.

The next line shows the same for another element:

wait.WaitUntil<AnotherElementType>(element2,

anyNameILike => anyNameILike.AnotherField == 1);

There are predefined WaitUntil’s for Elements and Behaviors, they do not need generics (<SomeElementType> or <AnotherElementType>), but they can work with Elements and Behaviors only.

Example with Element:

HtmlButton button = ElementService.Find<HtmlButton>().FindBy(How.Id, "button");

new SmartWait().WaitUntil(button, b => b.Displayed);

Example with Behavior:

HtmlTextField searchField = {…}

IPlaceholderBehavior placeholder = searchField.WithBehavior<IPlaceholderBehavior>();

new SmartWait().WaitUntil(placeholder,

p => p.GetPlaceholder() == "Expected Placeholder");

We can also specify different timeouts for SmartWait:

TimeSpan totalWaitTimeout = TimeSpan.FromMinutes(2);

TimeSpan retryInterval = TimeSpan.FromMilliseconds(500);

new SmartWait().SetTimeout(totalWaitTimeout)

.SetRetryInterval(retryInterval)

.WaitUntil(searchField, s => s.Displayed);

# Known issues

## Build errors

**Predefined type 'Microsoft.CSharp.RuntimeBinder.Binder' is not defined or imported**

**One or more types required to compile a dynamic expression cannot be found. Are you missing references to Microsoft.CSharp.dll and System.Core.dll?**

Go to Solution Explorer -> find your project -> right-click on References -> select Add Reference -> go to Assemblies tab -> find and add Microsoft.CSharp. Rebuild the project.