Bumblebee Documentation

Release 2.0.8-beta

Patrick Herrmann, John Hammerlund, and Todd Meinershagen

Mar 31, 2018

User Documentation

Bumblebee is a .NET layer on top of the Selenium browser automation framework that allows for the standardized creation of page objects, even for dynamic web pages. There are a few features that define Bumblebee's usability.

- Standardized UI interfaces
- Modular page objects
- IntelliSense-driven automations
- Test Framework independence
- Parallelization mindfulness
- Flexibility

Like other page object models, Bumblebee divides testing into two parts. The page objects model the subject of the testing, and the automation uses the page objects to tell the browser what to do.

Bumblebee standardizes the design of page objects and makes the automation scripting trivial. Develop page objects quickly by modeling which parts of the page can be interacted with. Then in the automation, use Intellisense to browse through the available options and build a script.

The basic idea behind Bumblebee is to break each page down into *Blocks* and *Elements*. Use the classes provided by Bumblebee to model your site into page objects that you can consume in your automation code. If page objects are well designed, writing the automation code should take no effort at all.

If you are new to Bumblebee, check out the *Getting Started* page first.

CHAPTER 1

Benefits

Bumblebee is a .NET layer on top of the Selenium browser automation framework that allows for the standardized creation of page objects, even for dynamic web pages. There are a few features that define Bumblebee's usability.

- Standardized UI interfaces
- Modular page objects
- IntelliSense-driven automations
- Test Framework independence
- Parallelization mindfulness
- Flexibility

1.1 Extensible Page Objects

In the typical page object model, a class is build to represent a page. Each method indicates an action that can be performed on the page. While this is helpful for writing automations, it is not as expressive as it could be.

Instead of typical page objects, Bumblebee uses blocks. A block is a specific area on a web page. This could be the entire page (like a page object) or a subset, like a tab menu, a sidebar, or a table. Each block is represented by a class that extends `Block`. Blocks can contain UI elements, like links, text fields, and check boxes. Blocks can also contain other blocks. A block's contents are represented as properties of that block.

This model allows us to represent our pages in a modular way, grouping common elements together and nesting things in a natural way.

1.2 Write Automations Using IntelliSense

In Bumblebee, we start off an automation by specifying at which page we'd like to begin. Doing so returns a block representing our current scope (which object our test is focusing on). That block then has properties (seen by IntelliSense) for each element contained within. After selecting which element we'd like to interact with, we can view its

methods. A check box, for example, will have methods `Check`, `Uncheck`, and `Toggle`. Each method performs the action returns the next block. From here the process repeats.

1.3 Interact With Standardized UI Elements

Bumblebee provides interfaces for the most common web UI elements, like links, buttons, alert dialogs, check boxes, select boxes, and text fields. On the web each of these standard elements can be implemented in a variety of ways. A web page might implement a select box by using the select tag, or by using a jQuery UI element, etc. In Bumblebee, an automation only ever reacts with the interface; an automation knows it's interacting with a select box, but it doesn't care what kind. When the element is created as a property of some block, it returns the interface but instantiates the actual implementation. Bumblebee comes with all the standard implementations, and allows the user to create their own implementations as well.

1.4 Test Framework Independence

Bumblebee can be used with any test framework, allowing you the freedom to set up your own project. You can put your page objects in one project, and put your automations in another, running them with whichever x-unit framework makes sense. Each browser session is instantiated with a driver environment, specifying how to create the driver for that particular session. Thus you can have multiple environments, such as a local environment (running on your local machine) and a grid environment (running on some remote grid). The parameters these take in (which browser, timeout specification, etc.) is entirely up to you.

1.5 Automation Flexibility

Often in a complicated web site, page objects are plagued by conditional specifications. For example, clicking a link leads to some specific page 99% of the time, but some other page under very specific circumstances. A page object has rigid methods, and can't support the edge cases as easily. With Bumblebee, interacting with elements typically allows the automator to tack a generic type onto the method, changing the block that is returned. This makes test cases extremely flexible, and takes some of the burden off of designing blocks for complicated sites.

CHAPTER 2

Getting Started

To get started with Bumblebee, let's do a simple example of logging in to a website and verifying that we have successfully logged in.

2.1 1. Install Libraries

In order to run the example, you will need to create a project in Visual Studio that is a class library. After the project is created, you will need to install the appropriate libraries. These should include a test library, Bumblebee, and any web drivers. You could also include any assertions libraries that you prefer.

For the purpose of this tutorial, we will install NUnit, Bumblebee, and the Internet Explorer web driver. You can do this by running the following commands at the package manager console or by selecting the packages via the *Tools/NuGet Package Manager/Manage NuGet Packages for Solution* dialog.

```
PM> Install-Package NUnit
PM> Install-Package Bumblebee.Automation
PM> Install-Package Selenium.WebDriver.IEDriver
```

2.2 2. Create Page/Blocks

The next step is to create the page/blocks that you want to use to represent the pages that you will interact with during the test automation. In this case, we would like to create a LoggedOutPage and LoggedInPage for the http://www.reddit.com site.

2.2.1 LoggedOutPage.cs

```
public class LoggedOutPage : WebBlock
{
  public LoggedOutPage(Session session) : base(session)
```

```
  {
  }

  public ITextField<LoggedOutPage> Email
  {
    get { return new TextField<LoggedOutPage>(this, By.Name("user")); }
  }

  public ITextField<LoggedOutPage> Password
  {
    get { return new TextField<LoggedOutPage>(this, By.Name("passwd")); }
  }

  public IClickable LoginButton
  {
    get { return new Clickable(this, By.TagName("button")); }
  }
}
```

2.2.2 LoggedInPage.cs

```
public class LoggedInPage : WebBlock
{
  public LoggedInPage(Session session) : base(session)
  {
    // Wait until we're logged in, then re-select the body to keep the DOM fresh
    Wait.Until(driver => driver.GetElement(By.CssSelector(".user a")));
    Tag = Session.Driver.GetElement(By.TagName("body"));
  }

  public IClickable<LoggedInPage> Profile
  {
    get { return new Clickable<LoggedInPage>(this, By.CssSelector(".user a")); }
  }

  public IClickable<LoggedOutPage> Logout
  {
    get { return new Clickable<LoggedOutPage>(this, By.LinkText("logout")); }
  }
}
```

2.3 3. Scripting the Test

After the page/blocks have been created, you can now reference them within a test.

```
[TestFixture]
public class LoginTests
{
    [Test]
    public void given_valid_credentials_when_logging_in_then_logged_in()
    {
        Threaded<Session>
          .With<InternetExplorer>()
```

```
            .NavigateTo<LoggedOutPage>("http://www.reddit.com")
            .Email.EnterText("bumblebeeexample")
            .Password.EnterText("123abc!!")
            .LoginButton.Click<LoggedInPage>()
            .VerifyPresenceOf("the logout link", By.CssSelector(".user a"));
    }

    ///The tear down operation is needed in case there is a failure during the test. ␣
↪The Session will need to be
    ///cleaned up.
    [TearDown]
    public void TearDown()
    {
        Threaded<Session>
          .End();
    }
}
```

Now that you have completed your first Bumblebee automation, it's time to explore the concepts in more detail.Good luck!

And let us know if you have questions or concerns here.

CHAPTER 3

Concepts

Bumblebee is a .NET layer on top of the Selenium browser automation framework that allows for the standardized creation of page objects, even for dynamic web pages.

Like other page object models, Bumblebee divides testing into two parts. The page objects model the subject of the testing, and the automation uses the page objects to tell the browser what to do.

Bumblebee standardizes the design of page objects and makes the automation scripting trivial. Develop page objects quickly by modeling which parts of the page can be interacted with. Then in the automation, use Intellisense to browse through the available options and build a script.

The basic idea behind Bumblebee is to break each page down into blocks and elements. Use the classes provided by Bumblebee to model your site into page objects. Consume page objects by writing your automation code. If page objects are well designed, writing the automation code should take no effort at all.

The following are concepts that you should be familiar with in Bumblebee:

- Blocks
- Elements
- Sessions
- Using Linq
- Conveniences

CHAPTER 4

Blocks

4.1 Overview

A block is an area on a web page. This could be an entire page, or a subset, like a tab menu, or a sidebar. Each block is represented by a class that extends *Block*. Blocks can contain child elements or child blocks. Each child is represented by a property in the parent block's class. Say we have some home page *HomePage*.

```
public class HomePage : Block
{
    public HomePage(Session session) : base(session)
    {
    }

    // Properties go here
}
```

BaseBlock here is a base type that is not provided by Bumblebee, but extends Block. It is the base block for this project. See the section on block scope for an explanation.

4.2 Block Scope

Each block is associated with an HTML tag (an element in the DOM). All selectors within the block search within the block by searching only within its corresponding tag. By default, the scope of a block is that of its parent class. To narrow the scope of the block, set the *Tag* property in the constructor of the block. For example, say the user search mechanism is on several different pages, but is always wrapped in a *div* with class "userSearch". We should create a block to represent it.

```
public class UserSearchBox : Block
{
    public UserSearchBox(Session session) : base(session)
    {
        Tag = GetElement(By.ClassName("userSearch"));
```

```
    }

    public ITextField<UserSearchBox> UsernameField
    {
        get { return new TextField<UserSearchBox>(this, By.Id("userSearchField")); }
    }

    public IClickable<UserSearchResultsBox> SearchButton
    {
        get { return new Clickable<UserSearchResultsBox>(this, By.Id("userSearchButton
↪")); }
    }
}
```

In the constructor, the scope is set to only search within the *div* in question. Because each constructor calls its base constructor first, scope narrowing cascades down from the base block. This raises the question, what is the default *Tag* selected from Block? *There actually isn't one*. There should only be one class that directly extends Block, which is to be used as your base block from then on. Lets create one.

```
public class BaseBlock : Block
{
    public BaseBlock(Session session) : base(session)
    {
        Tag = Session.Driver.GetElement(By.TagName("body"));
    }
}
```

This class will serve as the base for all of our blocks. We set the initial scope to be inside the body of the page here, but you can make it whatever you want. Notice we cannot use *GetElement*; *GetElement* searches within the current scope, which is not yet defined (we are currently defining it). Now we change our *UserSearchBox* class to extend the right block.

```
public class UserSearchBox : BaseBlock
{
    public UserSearchBox(Session session) : base(session)
    {
        Tag = GetElement(By.ClassName("userSearch"));
    }

    public ITextField<UserSearchBox> UsernameField
    {
        get { return new TextField<UserSearchBox>(this, By.Id("userSearchField")); }
    }

    public IClickable<UserSearchResultsBox> SearchButton
    {
        get { return new Clickable<UserSearchResultsBox>(this, By.Id("userSearchButton
↪")); }
    }
}
```

The *GetElement* call in the constructor searches within the current scope, which is the body of the page, as specified in the base block. The `By.Id("userSearchField")` selector in the *UsernameField* property will search within the current scope, an element with class "userSearch". Limited selectors to a scope often makes for simpler selectors.

Suppose there is a user search box on the home page. We can add one like so:

```
public class HomePage : BaseBlock
{
    public HomePage(Session session) : base(session)
    {
    }

    public UserSearchBox UserSearchBox
    {
        get { return new UserSearchBox(Session); }
    }

    ...
}
```

After we add the ProfileLinks property to a block called *UserSearchResultsBox*, we can do this:

```
Session.NavigateTo<HomePage>(url)
    .UserSearchBox
    .UsernameField.EnterText("Corey Trevor")
    .SearchButton.Click()
    .ProfileLinks.First().Click();
```

4.3 Specific Blocks

Child blocks are cool, and so is using linq to play with lists of elements. There's no reason we can't play with lists of blocks! Here's an example where we might do that.

Suppose our webpage has a table. Each row of the table represents some user, and has three things (bear with me):

- A link to the user's profile
- A check box indicating whether or not they are active
- A select box indicating their favorite color

We would like to be able to interact with any of these elements from our automation.

This block is different from any of the other blocks we've discussed thus far. There can be multiple instances on a page, so to distinguish we must pass it the web element we're talking about. For this type of block we use *SpecificBlock*. If I were to ask you for an instance of a block, and you had to ask "which one?", then that block should be modeled by a *SpecificBlock*.

```
public class UserTableRow : SpecificBlock
{
    public UserTableRow(Session session, IWebElement tag) : base(session, tag)
    {
    }

    public IClickable<ProfilePage> ProfileLink
    {
        get { return new Clickable<ProfilePage>(this, By.ClassName("profileLink")); }
    }

    public ICheckbox<UserTableRow> ActiveCheckbox
    {
        get { return new Checkbox<UserTableRow>(this, By.ClassName("activeCheckbox"));
↪ }
```

```
    }

    public ISelectBox<UserTableRow> FavoriteColorSelectBox
    {
        get { return new SelectBox<UserTableRow>(this, By.ClassName(
↪"favoriteColorSelect")); }
    }
}
```

Now we can set up the list of blocks.

```
public IEnumerable<UserTableRow> UserTableRows
{
    get
    {
        return GetElements(By.CssSelector(".userTable tr"))
            .Select(tr => new UserTableRow(Session, tr));
    }
}
```

Lets say for simplicity that this table is on the home page. Now we can write a test case like this:

```
Session.NavigateTo<HomePage>(url)
    .UserTableRows.First()
    .ActiveCheckbox.Check()
    .FavoriteColorSelectBox.Options.Random().Click()
    .ProfileLink.Click();
```

This gives the automation script the power to interact with anything on the page, using very little code.

Because specific blocks require an element to clarify which block you're talking about, specific blocks cannot be the target of elements (you should never write `IClickable<UserTableRow>` because it wouldn't know which row to give you or how). The exception is from inside the specific block itself (inside *UserTableRow* we have `ISelectBox<UserTableRow>`), in which case Bumblebee understands that the only element you could logically be referring to is the parent one (itself).

4.4 Current Block

To continue an automation without continuing one long expression, use *Session.CurrentBlock* and pass the type of the page you're on. This is especially useful when setting up test cases. A suite of tests might start out the same; they go to the site and log in. Then the individual test cases can pick up from there with `Session.CurrentBlock<LoggedInPage>...`.

Similarly, if you want to switch focus to a different block you can do it inline with `ScopeTo<SomePage>`.

CHAPTER 5

Elements

An element is a user interface item. Examples include links, buttons, checkboxes, select boxes, etc. Each type of element is represented by an interface. For example IClickable represents both buttons and links (with a single method, "Click").

Each element on a page gets represented by a property. The most important part of the property is the return type. The type returned should be the interface of the element, for example *ISelectBox*. This is how the user (the person writing the automation test case) can interact with the element.

5.1 The Interface

The generic type parameter of the return type is where the scope returns to after the element is interacted with. For elements that generally do not change the page, the generic type is typically the type of the parent block (the block of which the element is a property). Suppose there is a select box on our home page for the user to choose their favorite color. We define the property like this:

```
public ISelectBox<HomePage> FavoriteColorSelectBox { ... }
```

For elements that generally do change the state of the page, the type parameter is the type of the block that is led to. For example, say we have a link to an about page *AboutPage*.

```
public IClickable<AboutPage> AboutLink { ... }
```

Sometimes, however, performing an action could lead to several different places on the site. For example, say we have a "Log In" button. Clicking this button could lead to the profile page *ProfilePage* or stay on the home page with an invalid login message. For elements like this there is no default block where we end up, so we'd like the user to specify where they think it should end up in the context of the automation. To do so, we just leave off the generic type parameter:

```
public IClickable LogInButton { ... }
```

Notice there is no generic type parameter after *IClickable*; we don't know where it will go so we don't specify ahead of time.

If there's a link on a page that leads to the same block 99% of the time, but leads somewhere else in rare cases (for example an error popup or a log in page for privileged links), you should still return an *Clickable* leading to the most common return case. In your test case you can still override the default return type by specifying a generic parameter. In other words, giving your interface a generic type sets a default and allows the automator to leave it off.

5.2 The Implementation

To implement the property, we must return a concrete type. For most cases, each interface's corresponding implementation should suffice (Use *Clickable* for *IClickable*). We diverge from this pattern when a site contains custom UI elements (like jQuery select boxes and such). If the element on the page fits the interface, but is implemented differently, make a custom implementation and use that.

Element implementations take two parameters. The first is the parent block of the element, which it uses for scope. Just pass *this*, as we are calling from the parent block. The second is the selector for the element (a Selenium *By* object). Alternatively, you can pass the *IWebElement* representing the element, which is often useful.

Here are the properties above implemented fully:

```
public ISelectBox<HomePage> FavoriteColorSelectBox
{
    get { return new SelectBox<HomePage>(this, By.Id("favoriteColor")); }
}

public IClickable<AboutPage> AboutLink
{
    get { return new Clickable<AboutPage>(this, By.Id("aboutLink")); }
}

public IClickable LogInButton
{
    get { return new Clickable(this, By.Id("logInButton")); }
}
```

and in the context of our *HomePage* class with some login elements added:

```
public class HomePage : Page
{
    public HomePage(Session session) : base(session)
    {
    }

    public ISelectBox<HomePage> FavoriteColorSelectBox
    {
        get { return new SelectBox<HomePage>(this, By.Id("favoriteColor")); }
    }

    public IClickable<AboutPage> AboutLink
    {
        get { return new Clickable<AboutPage>(this, By.Id("aboutLink")); }
    }

    public IClickable LogInButton
    {
        get { return new Clickable(this, By.Id("logInButton")); }
    }
```

```
    public ITextField<HomePage> UsernameField
    {
        get { return new TextField<HomePage>(this, By.Id("usernameField")); }
    }

    public ITextField<HomePage> PasswordField
    {
        get { return new TextField<HomePage>(this, By.Id("passwordField")); }
    }
}
```

5.3 Text Fields

A TextField class is typically is used to represent an input element of text, date, or numeric type or a textarea element.

5.3.1 AppendText(string text)

This method allows users to add text to the end of any existing text within a text field.

5.3.2 EnterText(string text)

This method clears any existing text for a text field and then adds the text to the field.

5.3.3 Press(Key key)

This method allows users to press single keys in a text field including key combinations. For example,

```
myPage.Press(Key.A);
myPage.Press(Key.Control + Key.C);
myPage.Press(Key.Control + Key.Alt + Key.Delete);
```

5.4 Tables

Tables are meant to be a simple set of abstractions for tables within your HTML. If your application is using the basic HTML 4.0 tables as shown below:

```
<table id="myTable">
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Todd</td>
      <td>Meinershagen</td>
    </tr>
  </tbody>
```

```
  <tfoot>
    <tr>
      <td>Number of People:</td>
      <td>1</td>
    </tr>
  </tfoot>
</table>
```

You can then represent as an ITable with the Table implementation like so:

```
public class TablePage : WebBlock
{
  public ITable MyTable
  {
    get { return new Table(this, By.Id("myTable"));
  }
}

//Using the Page in Test Scripts
NavigateTo<TablePage>("http://urltopage.com")
  .Table
  .VerifyThat(t => t.Headers[0].Should().Be("First Name"))
  .VerifyThat(t => t.Rows.First()["Last Name"].Should().Be("Meinershagen"))
  .VerifyThat(t => t.Footer[1].Should().Be("1");
```

CHAPTER 6

Sessions

6.1 Constructing Sessions

Before we can use our page/block objects, we need to set up a browser *Session*. To construct one, you will need to provide a driver environment, which is any class that implements the *IDriverEnvironment* interface. You can learn more about driver environments here.

For the example below, we are using a built-in environment called *InternetExplorer*.

```
var environment = new InternetExplorer();
var session = new Session(environment);
```

You can also assemble *Session* with one line, as long as the *IDriverEnvironment* you have chosen to use has a default constructor.

```
var session = new Session<InternetExplorer>();
```

6.2 Session Fixtures

In order to make this reusable in your preferred test framework, you would want to create a base test fixture with this variable as protected. Any derived classes could then reference this variable. The example below is for NUnit.

```
[TestFixture]
public abstract class SessionFixture
{
  protected Session Session { get; private set; }

  [SetUp]
  public void SetUp()
  {
    Session = new Session<InternetExplorer>();
  }
```

```
  [TearDown]
  public void TearDown()
  {
    Session.End();
  }
}

[TestFixture]
public class LoginPageTests : SessionFixture
{
  [Test]
  public void given_invalid_login_when_logging_in_should_display_error()
  {
    Session
      .NavigateTo<LoginPage>("http://someuri.org/login")
      .VerifyPresence(By.Id("invalidLoginMessage"));
  }
}
```

6.3 Thread-Safe Sessions

An alternative to creating a common base test fixture is to use a built-in, thread-safe method for establishing a *Session* that can be used across multiple methods within a test fixture or even across multiple fixtures on one thread without having to first construct it.

An example is below.

```
[TestFixture]
public class LoginPageTests
{
  [Test]
  public void given_invalid_login_when_logging_in_should_display_error()
  {
    Threaded<Session>
      .With<InternetExplorer>()
      .NavigateTo<LoginPage>("http://someuri.org/login")
      .VerifyPresence(By.Id("invalidLoginMessage"));
  }

  [TearDown]
  public void TearDown()
  {
    Threaded<Session>
      .End();
  }
}
```

You can keep referencing the same *Session* in each and every method. And if you are leveraging a test framework that spins off multiple threads running your tests, you can be assured that each *Session* is in it's own thread and safe from harm.

In some cases, you will want to use a driver environment that does not have a default public constructor. In those cases, you can pass in an instance to the With() method.

```
[Test]
public void given_invalid_login_when_logging_in_should_display_error()
{
  Threaded<Session>
    .With(new InternetExplorer(TimeSpan.FromSeconds(5)))
    .NavigateTo<LoginPage>("http://someuri.org/login")
    .VerifyPresence(By.Id("invalidLoginMessage"));;
}
```

6.4 Screen Capture

TODO

CHAPTER 7

Settings

TODO

CHAPTER 8

Driver Environments

A driver environment is responsible for creating a web driver instance based on some context. That context is up to you. By encapsulating this set up in one class, you can reuse this across multiple Sessions within your test suite.

Here's an example of a driver environment that constructs an Internet Explorer web driver that maximizes the window and will implicitly wait for up to 5 seconds when trying to find web elements.

```
public class LocalEnvironment : IDriverEnvironment
{
  public IWebDriver CreateWebDriver()
  {
    var driver = new InternetExplorer();
    driver.Manage().Window.Maximize();
    driver.Manage().Timeouts().ImplicitlyWait(TimeSpan.FromSeconds(5));
    return driver;
  }
}

var session = new Session<LocalEnvironment>();
```

8.1 Built-In Environments

The framework includes a set of built-in driver environments that you can leverage immediately within your test fixtures.

- Chrome
- Firefox
- InternetExplorer
- PhantomJS

8.2 Custom Environments

You can create your own environments by either implementing the IDriverElement interface or inheriting from the base SimpleDriverEnvironment class.

CHAPTER 9

Verification

When writing automated test cases, you often want to verify that particular condition is true. When it isn't, you want a meaningful exception to be thrown, failing the test case. Using Bumblebee, these verifications are performed using extension methods.

9.1 Verify

The most versatile form of verification is the `Verify` method. This method can be called at any time during your test case. It's parameter is a predicate; a function that returns true or false. The function you pass to `Verify` takes one parameter, the object that called verify. The function is typically created anonymously with a lambda expression.

In addition, `Verify` takes whatever object called it and passes it into your function. After performing the verification, it returns the object that called it. This is what allows you to chain verifications together.

Here is an example.

```
var s = "foo";
s
  .Verify(x => x.Length == 3)
  .Verify(x => x.EndsWith("o"));
```

The `x` variable is simply our name for the parameter that the predicate accepts. A lambda expression is really just an anonymous form of a function that takes in parameters and returns values.

For example, you could express the previous code as follows:

```
public void Test()
{
  var s = "foo";
  s
    .Verify(LengthIsThree)
    .Verify(EndsWithO);
}
```

```
public bool LengthIsThree(string x)
{
  return x.Length == 3;
}

public bool EndsWithO(string x)
{
  return x.EndsWith("o");
}
```

Using this technique, you could make your code look more readable as well as avoid the use of lambda expressions.

9.2 Adding Readability

To make our test cases more readable, we can add a string argument describing what we are verifying, like so:

```
var s = "foo";
s
.Verify("that the length is 3", x => x.Length == 3)
.Verify("that the string ends with 'o'", x => x.EndsWith("o"));
```

Adding these strings makes it easy to read through the verifications, and adds a level of self-documentation. In addition, when a verification fails, the failure message will be representative of what happened. Instead of "Unable to verify custom verification", you will get a much more helpful error message, "Unable to verify that the length is 3".

9.3 VerifyThat

Another way to add readability to your verifications is to leverage your own preferred assertion library. The `VerifyThat` method takes whatever object called it and passes it into your assertion action. Below is an example using assertions from the NUnit and FluentAssertions libraries.

```
var s = "foo";
s
.VerifyThat(x => Assert.IsTrue(x.StartsWith("f")) //nunit
.VerifyThat(x => x.Should().Contain("o"));        //fluent assertions
```

9.4 Verifying Presence

There are some additional verification extension methods that handle more specific situations. Say we wanted to make sure that an error message has appeared on the site, with an ID of "errorMessage". We can do so by passing the selector.

```
...
.VerifyPresence(By.Id("errorMessage"));
```

Like with `Verify`, we can add a string as the first argument to describe in English what we are verifying.

```
...
.VerifyPresenceOf("an error message", By.Id("errorMessage"));
```

Similarly we could make sure that there is no such error message:

```
...
.VerifyAbsence(By.Id("errorMessage"))
.VerifyAbsenceOf("an error message", By.Id("errorMessage"));
```

9.5 Convenience Verifications

Bumblebee comes with some convenience verification methods that only work on some types. For example, here are some verifications that can only be performed on things with text, that is, elements that implement the `IHasText` interface.

```
.SomeTextField
.VerifyText("Verify this text matches exactly")
.VerifyTextContains("Verify the element's text contains this text")
.VerifyTextMismatch("Verify the element's text does not match this text")
...
```

These are merely convenience methods. For example, the examples above could more verbosely be written like this:

```
.SomeTextField
.Verify(hasText => hasText.Text == "Verify this text matches exactly")
.Verify(hasText => hasText.Text.Contains("Verify the element's text contains this text
↪"))
.Verify(hasText => hasText.Text != "Verify the element's text does not match this text
↪")
...
```

When more specific verifications are required, you can also fall back to using lambda expressions. If you find yourself using very similar lambda expressions frequently, it would be a good idea to make your own verification method. Look at the Bumblebee source at *Verifications* for examples.

CHAPTER 10

Using Linq

The *Options* property in the first example above returns an *IEnumerable* of options. This allows the automator to decide which option to act on however they like. Using linq we get many methods for free. Here are some examples of valid selections:

```
Options.First().Click();
```

Clicks the first option.

```
Options.Last().Click();
```

Clicks the last option.

```
Options.ElementAt(3).Click();
```

Clicks the fourth option.

```
Options.WithText("Green").Single().Click();
```

Clicks the option with text "Green". Throws an exception if there aren't any or if there are more than one.

```
Options.WithText("Green").First().Click();
```

Clicks the *first* option with text "Green". Throws an exception if there aren't any, but not if there are multiple.

```
Options.Skip(1).Random().Click();
```

Clicks any but the first option at random.

```
Options.Reverse().Skip(1).Random().Click();
```

Clicks any but the last option at random.

```
Options.Where(opt => opt.Text.EndsWith("e")).First().Click();
```

Clicks the first options whose text ends with "e".

```
Options.Unselected().Random().Click();
```

Click a random option excluding the one already selected.

Note that *Random*, *WithText*, and *Unselected* are not part of linq, but come with Bumblebee.

We can use this pattern whenever we have a collection of similar elements. Suppose we have a user search page on our site. The search results come back as a list of user profile links. We can easily model all of these links with a single property:

```
public IEnumerable<IClickable<ProfilePage>> ProfileLinks
{
    get
    {
        return GetElements(By.ClassName("profileLink")).Select(link => new Clickable
↪<ProfilePage>(this, link));
    }
}
```

Now in our automations we can type

```
ProfileLinks.First().Click();
```

along with all the other selectors.

CHAPTER 11

Conveniences

11.1 Pause

Call *Pause(int miliseconds)* at any point in your automation and execution will pause. Scope will be returned exactly where it was.

11.2 Hover

To hover over any element, call *Hover(int miliseconds)* to hover the mouse over that element for the specified number of seconds. Like when pausing, scope will be returned to exactly where it was.

11.3 Verification

Perform verifications from within your automation expression. Scope returns to where it was before the call. There are specific verification methods for verifying the presence or absence of an element, verifying the text of an element, and verifying the selected/unselected status of an option. For a greater degree of flexibility, you can call *Verify* at any point in your test case, and give a predicate to verify in the form of a lambda expression. The parameter of the lambda expression is the calling object.

The example below verifies that a select box has five options and then clicks the last.

```
...
.SomeSelectBox.Options
.Verify(opts => opts.Count() == 5)
.Last().Click()
...
```

More information about verification

11.4 Drag and Drop

Selenium supports dragging elements onto other elements, as well as dragging elements by a certain offset. Lets say we have a web page which has a visual calendar in which appointments can be dragged from one time to another. Lets say the page object, `AppointmentCalendarPage`, has properties `Appointments` and `TimeSlots`, which each return `IEnumerable<IClickable<AppointmentCalendarPage>>` so that the appointments and time slots can be filtered through using LINQ. The following automation drags the first appointment into the last time slot, and then drags the last appointment up 50 pixels. The second drag anticipates an alert dialog, and dismisses it.

```
...
.AppointmentCalendarPage
.Drag(acp => acp.Appointments.First()).AndDrop(acp => acp.TimeSlots.Last())
.Drag(acp => acp.Appointments.Last()).AndDrop<AlertDialog>(0, -50)
.Dismiss<AppointmentCalendarPage>();
```

11.5 Debug printing

Debug by printing information within automations. Call *DebugPrint* to print the current object, or pass a function to operate on the object first. The example below clicks the last option of a select box, but first prints the first option's text.

```
...
.SomeSelectBox.Options
.DebugPrint(opts => opts.First().Text)
.Last().Click()
...
```

If your lambda returns an `IEnumerable` of some kind, `DebugPrint` will print each value. For example, you can instead print all of the select box options like so:

```
...
.SomeSelectBox.Options
.DebugPrint(opts => opts.Select(opt => opt.Text))
.Last().Click()
...
```

11.6 Frames

Frames are easy! They are just blocks. Other blocks can extend them, which causes the frame to be selected before the child block's constructor is even executed. Is the whole site content in a frame? Select the frame in your base block and forget about it.

After selecting a frame in the constructor in a frame block, don't forgot to set the the *Tag* element. Any old elements you had selected will be stale after switching frames. Throw in a `Tag = Session.Driver.GetElement(By.TagName("body"));` and you should be all set to continue nesting blocks.

11.7 Alerts

Ordinarily if an alert appears you will wind up with an error indicating that you were blocked by a modal dialog. Sometimes, however, you expect an alert to appear during your automation and you'd like to deal with it. For these

situations there is a block that represents alerts.

Say we have a link that deletes a user's profile. Clicking it brings up an alert prompting whether or not you are sure. Canceling stays on the profile page, and accepting goes to the home page. We should set up the link as an `IClickable<IAlertDialog>`. Here are some possible automations:

```
...
.DeleteProfileLink.Click()
.Accept<HomePage>()
...
```

```
...
.DeleteProfileLink.Click()
.Dismiss<ProfilePage>()
...
```

Alerts can sometimes appear due to very convoluted conditions. In your automation, if you expect an alert to appear, you can call `Click<AlertDialog>()`. *AlertDialog* is Bumblebee's *IAlertDialog* implementation representing popup alerts. If your site uses custom alerts, like lightboxes, make a block that implements *IAlertDialog* and use that instead.

11.8 Storing temporary data

Often during a test it is necessary to verify that two things match. For example, say you change the user's favorite color by randomly selecting a different option in their favorite color select box. Then you view the home page and you want to confirm that it lists their new favorite color. To do this, we need to store the text of the randomly selected option until it's time to verify it. For this we use the *Store* extension method.

Here's the code first:

```
string newFavoriteColor;

Session.NavigateTo<HomePage>(url)
    .UsernameField.EnterText("randylahey@sunnyvale.org")
    .PasswordField.EnterText("password1234")
    .LogInButton.Click<ProfilePage>()
    .FavoriteColorSelectBox.Options.Random()
    .Store(out newFavoriteColor, opt => opt.Text)
    .Click()
    .HomePageLink.Click()
    .Verify("the text has updated", page => page.FavoriteColor == newFavoriteColor);
```

First we create a variable in which to store our temporary data, *newFavoriteColor*. When we select our random option, before we click it, we can call *Store* to store information about it. Store takes our variable as an out parameter, which allows the extension method to assign to it. The second parameter is a function acting on the calling object, which in this case is the random option. We return the option's text for storage into the variable.

On the home page, the *FavoriteColor* property refers to some text on the home page. To add a property like this to a block, make a string property with a getter that finds the text on the page and returns it.

If you need to compare two stored values, use *VerifyEquality* or *VerifyInequality* which take any two objects.

11.9 Playing Sound

This last one started off as a joke, but the more I thought about the better the idea seemed. If you've written many automated tests before, you'll understand the following situation.

You run an automation with the goal of observing or debugging a certain procedure. You wait twenty or thirty seconds while the test case is doing its thing. You space out for not five seconds. You look back at the screen and the test is over; you missed the part you actually wanted to see. You rerun the test and repeat the whole process again, getting more and more frustrated with yourself.

Bumblebee's "solution" is rather ridiculous: play a "dinging" sound right before the part you care about. When debugging, place `PlaySound()` right before the interesting part of your test and Bumblebee will play a system sound. You can optionally pass it a number of seconds to pause, to give you time to switch to the window or what have you.

CHAPTER 12

Dependencies