System Manual

Javier Pascual and Sam Pham

April 25, 2017

1 Intro

The aim of this manual is to outline the basic structure of the code the LSCK is built from. We will talk about three main subsections, which represent the complete design of the app and together allow the user to perform all actions LSCK provides. They are:

1. UI Classes
2. Backend
3. Helper classes

2 Dependencies

The extension has a number of dependences in the form of NuGet packages that they are as follows:

- Renci.SshNet
- Newtonsoft.Json
- WPFToolkit

3 UI Classes

In terms of the UI, there are 3 main tool windows, which have an XAML file for their design, and an xaml.cs file which controls the UI functions. The three tool windows are the following:

1. LSCK tool window
2. Structure tool window
3. Site Preview tool windo

The XAML is responsible for positioning and implementing all buttons, listboxes, dropdowns etc. which the user can visualize when the extension is installed. The following is a small snippet of code extracted from the XAML file of the main LSCK UI window. It is responsible for displaying the sections already created by the user and the button which enables them to create a new one.

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Center" >
    <ComboBox x:Name="comboSectionsCode" SelectionChanged="comboSectionsCode_SelectionChanged"
    HorizontalAlignment="Center" Width="167"  Foreground="#FF273380" BorderBrush="#FF0F7C45"
    FontFamily="Sitka Small" Height="20.8"/>

    <Button x:Name="createSection" Click="createSection_Click" Content="Create" Height="20.8"
    Width="85" Background="#314765" BorderBrush="{x:Null}" Margin="20,0,0,0" Foreground="White"
    BorderThickness="0"/>
</StackPanel>
```

3.1 LSCK tool window

The LSCK tool window is the center of the application. It represents the main UI of the system, and through XAML code (like the above example), it calls UI methods which in turn make use of the back-end classes to perform certain actions, like generate a website or modify a JSON file. Most of the methods in the xaml.cs of this class are button listeners, listbox listeners, click events, etc. For instance:

```
private void addButton_Click(object sender, RoutedEventArgs e)
{
  var selection = (TextSelection)dte2.ActiveDocument.Selection;
  string comment = textBox.Text;
  string code = selection.Text;
  string language = getConvertedLang();
  fjController.InsertSnippet(comboSectionsCode.SelectedValue.ToString(),language,comment,code);
  System.Windows.MessageBox.Show("Snippet added successfully");
}
```

The above code is really self-explanatory, it is responsible of performing a certain action for the click of a button. In this case, it is used for the automatic addition of code function that LSCK provides. It reads the selection of code in the Visual Studio IDE and uses the back-end singleton class "FJController" to create a new snippet file. With similar code to this throughout the class, the LSCK tool window allos the user to add files to their website, automatically add commented code, customize the display of the website and upload to the server.

Another very important function this class performs is testing to see when a new solution has been open (to update the UI with relevant changes), or checking to see if no solution is open (to disable the window). All the checking is done in the Bridge class, which will be mentioned later, but the LSCK tool window is responsible for creating a thread which runs in the background. Once the thread has been created and started, the following code runs (with small intermittent sleeps):

```
public void checkDTEChange()
{
    while (true)
    {
        System.Threading.Thread.Sleep(500);
        this.Dispatcher.Invoke(new Action(() => CheckDir()));
        if (state != 0)
        {
            this.Dispatcher.Invoke(new Action(() => resetControl()));
        }
    }
}
```

This tool window is the only one directly accessible by the user when the VSIX package is installed, as other tool windows can only be opened directly from this one (through button click events).

3.2 Structure tool window

The structure tool window can be opened through a button click from the LSCK tool window. Essentially, it has an identical functioning to the latter, as an XAML file is used to position elements like buttons and arrows, and listeners on these UI elements call events which are used to modify the structure of the website and the sections created by the user.

Both the Structure and the LSCK tool windows make use of an update method which takes in an integer value and updates the UI in one way or another according to the value passed. For instance, in this tool window, a call of "update(0)" will run the following:

```
case 0:
    List<string> pageNames = fjController.GetPageTitles();
    comboPages.Items.Clear();
    foreach (string pageName in pageNames)
    {
        comboPages.Items.Add(pageName);
    }
    break;
```

3.3 Site Preview tool window

Finally, we have the Site Preview tool window which will open up when the user clicks either the preview website or the upload button in the LSCK tool window.

This window is the most basic one, as can be seen from the XAML. It acts as a browser, and will display the contents of the generated HTML file by the extension. Once opened for the first time, a thread will run in the background and check for changes in the website, refreshing the site as needed. This again, is done with help of the Bridge class. The Refresh method is the following:

```
public void Refresh()
{
    if (Bridge.fjController != null)
    {
        string homepage = Bridge.fjController.GetPageTitles()[0];
        if (File.Exists(Bridge.solutionDir + "/generatedWebsite/" + homepage + ".html"))
        {
            Browser.Navigate(new Uri(String.Format("file:///{0}/generatedWebsite/{1}.html",
            Bridge.solutionDir, homepage)));
        }
    }
}
```

4 Back-end Classes

The back-end of LSCK is made of a number of different sections and classes, each with different functionality. They are as follows:

- Data Structure and Storage – FJController, JSON and FileHandler
- Uploading and setting up VM – SSH
- Creating portfolio – WebsiteGenerator

Because this is the back-end, you could potentially redevelop the front-end and use it openly. In terms of the data structure, all can be controlled by the FJController. It holds the top level functions such as creating or editing a snippet or section, getting data to send of specific sections. All functions from FJController contain descriptions which may help you understand their functionalities. What is meant by top-level is that it doesn't know the data structure, it's been abstracted but it tells the FileHandler and JSON what to do.

On the other hand, FileHandler and JSON are the lower level classes which is where you may want to edit the data structure, such as adding a new setting or customization to save.

4.1 Data Structure

JSON

Since most of the data is in the JSON data structure, this is by far the more important class. The overall workings of this class are:

1. When instantiated the JSON file is read and data is copied into the object from the JSON file.
2. When get methods are called, values are read from the object.

3. When set methods are called, values are changed in the object but then also written to the files

The data from the JSON file is only read once. It is when the object is instantiated. So if you want to read of a new file, you have to recreate the object. When you have edited the JSON data and you wish to save, just run the WriteJSON() function. You will notice that at end of nearly every method JSON has this function.

```
public void NullPage(string sectionName)
{
    int sectionIndex = getSectionIndex(sectionName);
    for (int x = 0; x < JSONFile.sections.Count; x++)
    {
        if (JSONFile.sections[x].page == JSONFile.sections[sectionIndex].page &&
        JSONFile.sections[x].position > JSONFile.sections[sectionIndex].position)
        {
            JSONFile.sections[x].position--;
        }
    }
    JSONFile.sections[sectionIndex].page = null;
    JSONFile.sections[sectionIndex].position = 0;
    WriteJSON();
}

public void InsertPage(string pageName)
{
    JSONFile.page_titles.Add(pageName);
    WriteJSON();
}
```

There are also three other private classes in JSON which are RootJSONObject, Section and Snippet. These are for the structure of the JSON. RootJSONObject is the top level. Then RootJSONObject has a list of Sections which then has a list of Snippets. If you wish to change the data structure as in add or remove certain values, change them here. Be sure to also change the Constructor after.

```
private class Snippet
{
    public int position { get; set; }
    public string language { get; set; }
    public string comment { get; set; }
}

private class Section
{
    public string sectionName { get; set; }
    public string page { get; set; }
    public int position { get; set; }
    public List<Snippet> snippets { get; set; }
}
```

FileHandler

FileHandler handles files that have been added by the user and files which hold code. This class was created as you can't actually store files in JSON. This class has similar methods to JSON but rather than manipulating JSON data, it manipulates files that have saved.

Unlike the JSON class, the contents of the files are read into the extension when needed rather than when the object is instantiated. This is because files only hold the content and not portfolio structure and so it's not necessary to do so.

4.2 Uploading and Setting up VM

The SSH class is made up of two methods. CreateConnectionInfo() which based on the input parameters will make the create details for the NuGet package SSH.Net to be able to connect to the server. It will then us UploadWebsite() to installed the necessary packages such as Apache if the virtual machine doesn't have it as well as uploading the website.

With CreateConnectionInfo, there is just a case statement for the different authentication methods such as password or PEM. You can just add more cases for other types of authentication. Though it is assuming that you will need an IP address and a username.

4.3 Creating the Portfolio

Generation of the website is done through the WebsiteGenerator class. The class was intended to be built in a different manner to generate the website, so there is a method called GenerateWebsite(). All you need do is add a parameter and a case statement. So far all it does it use the generateHTML() method. This method calls a number of different methods that generate different parts of the HTML page, such as the header, the body, the javaScript, the footer, etc.

```
public string GenerateHTML(string pageTitle)
{
    var htmlCL = new List<string>(); //HTMLContentList

    List<Section> page = fjController.GetPage(pageTitle);
    List<Snippet> pageSnippets = fjController.PageSnippetsOnly(page);

    htmlCL.Add(GenerateHead(fjController.GetTitle()));
    htmlCL.Add(GenerateBody(page, fjController.GetTitle(), pageTitle));
    htmlCL.Add(GenerateAceScript(pageSnippets, fjController.GetAceTheme()));
    htmlCL.Add(GenerateFooter());

    string htmlContent = string.Join("\n", htmlCL.ToArray());
    return htmlContent;
}
```

Adding the ability to use PHP should not be too difficult.

5 Helper classes

Our extension makes use of two helper classes (both static) which are used by the UI to perform certain actions and to serve as a bridge between the different tool windows that conform the user interface. These classes are the Bridge, and the Extractor class.

5.1 Bridge

The Bridge class has the main functionality of acting as a link between all the tool windows. It acts as a global variable container which can be modified by the all other classes. One of its main purposes is checking for a new solution. As mentioned before, the LSCK tool window creates a subthread which will run methods from this class. The methods are used to check if a new solution is open or if no solution is currently open.

An example section of the CheckDir method is:

```
public static void CheckDir()
{
    if (dte2 != null && solutionDir != null)
    {
        try
        {
            string newSolutionDir = Path.GetDirectoryName(dte2.Solution.FullName);
            if (solutionDir != newSolutionDir)
            {
                System.Windows.MessageBox.Show(newSolutionDir);
                solutionDir = newSolutionDir;
                fileDir = solutionDir + @"\LSCK Data";
                fjController.Reset();
                state = 1;
            }
        }
        ...
        ...
```

This part of the method acts when a solution directory has not yet been set (in other words, Visual Studio was open initially with no solution open). The method will get the new solution path, reset the FJController singleton, and modify the state variable within the Bridge class, which can then be used by the LSCK tool window to understand that changes must be made to the UI.

Another important functionality of the Bridge class is to serve as a communication point between the website generation methods of the LSCK window and the refreshing of the site preview.

5.2 Extractor

The extractor class is responsible for the automatic addition of code to the website. It has a main method called FindFiles which will browse through the entire open solution for files, and then for each file found will look for a specific character sequence.

```
foreach (string foundFile in foundFiles)
{
    if (extensions.Any(x => foundFile.EndsWith(x, StringComparison.Ordinal)))
    {
        string extension = Path.GetExtension(foundFile);
        Tuple<string,string> langProperties = getLanguageProperties(extension);
        FindSnippets(foundFile,langProperties.Item1,key,langProperties.Item2);
    }
}
```

This is a part of the FindFiles method. As we can see, for every found file in the solution, another method called FindSnippets will be called which will find as many snippets as have been created (in the manner supported by LSCK) and add them to the website.

The FindFiles method is called from the LSCK tool window through the Auto add button in the code addition tab of the window.