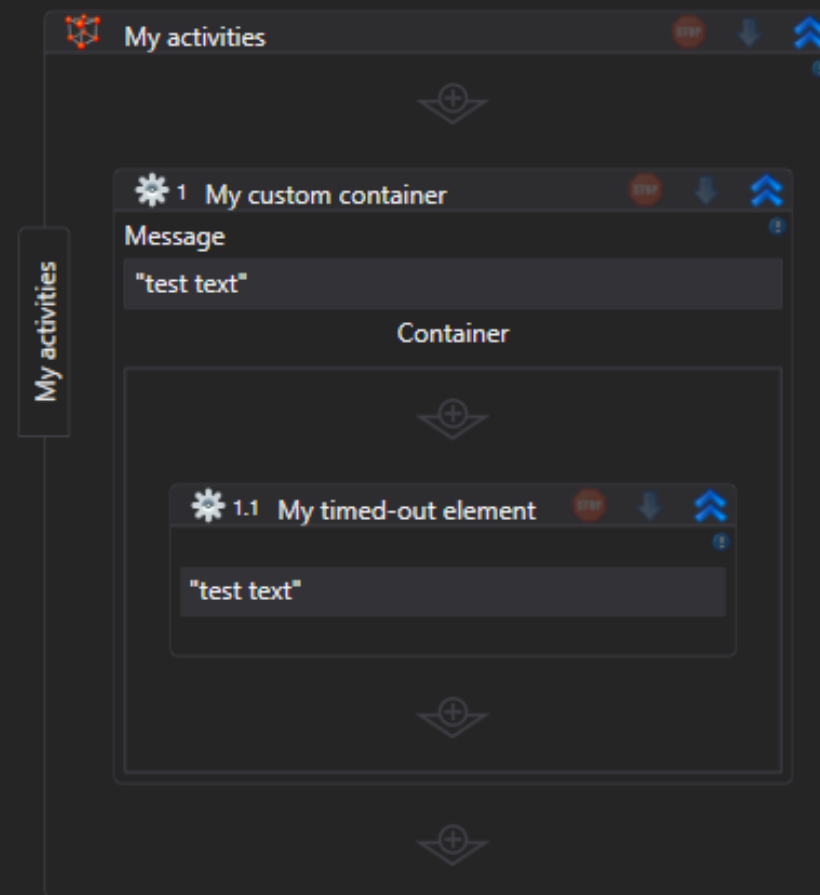




Создание элементов для Primo Studio

Команда миграции и внедрения
бизнес процессов

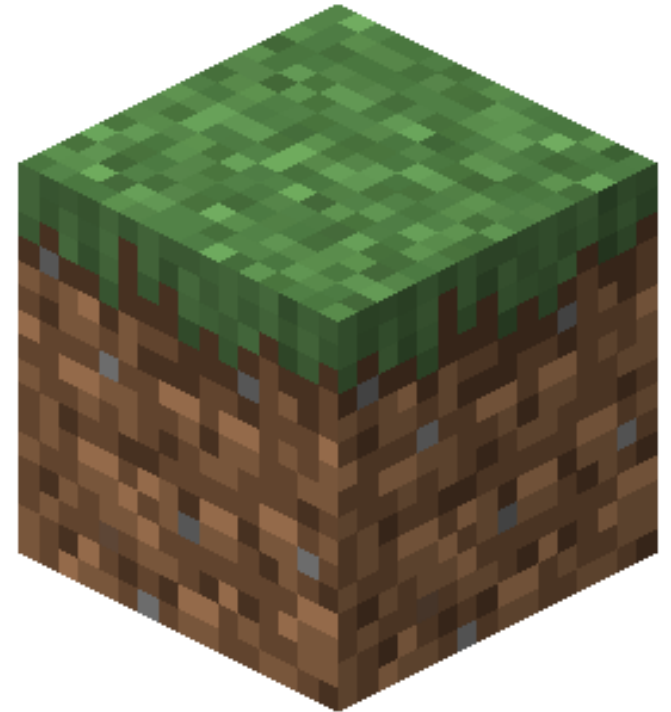


Об элементах Primo Studio

Элементы – это *строительные блоки* автоматизируемого процесса.

Каждый *элемент* это обособленное действие, которое в комбинации с другими элементами формирует необходимый алгоритм (workflow).

Primo Studio включает в себя набор готовых элементов, который может быть расширен с помощью сторонних NuGet-пакетов.



Зачем создавать свои элементы?

Primo RPA – молодая, развивающаяся платформа. Несмотря на обширный и постоянно пополняемый набор готовых элементов, в **Studio** может не оказаться подходящего элемента для решения специфичной задачи.

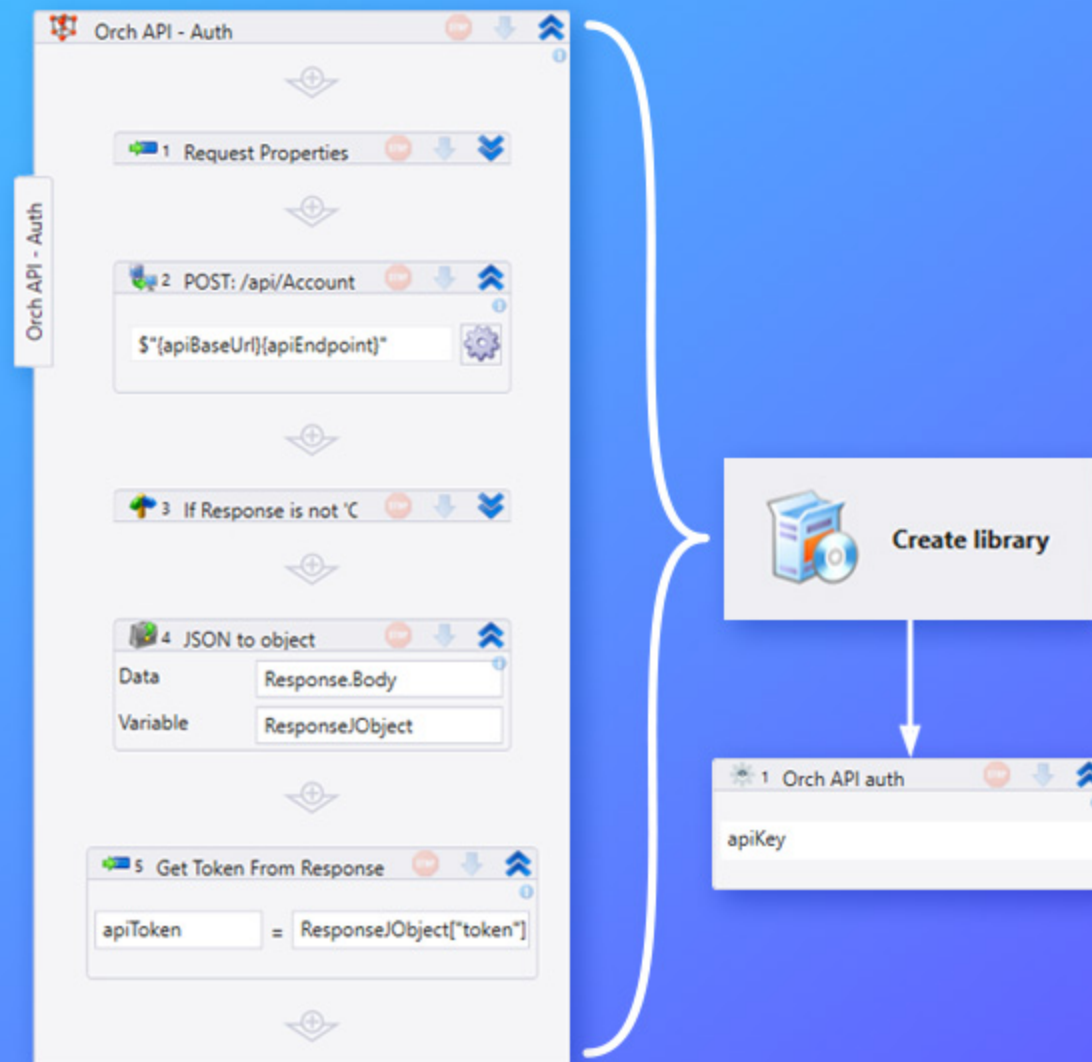
В такой ситуации может понадобиться создать свой собственный элемент, содержащий логику и GUI для решения ваших задач.

Как создать свой элемент?

Для создания элемента Primo Studio можно воспользоваться одним из двух вариантов:

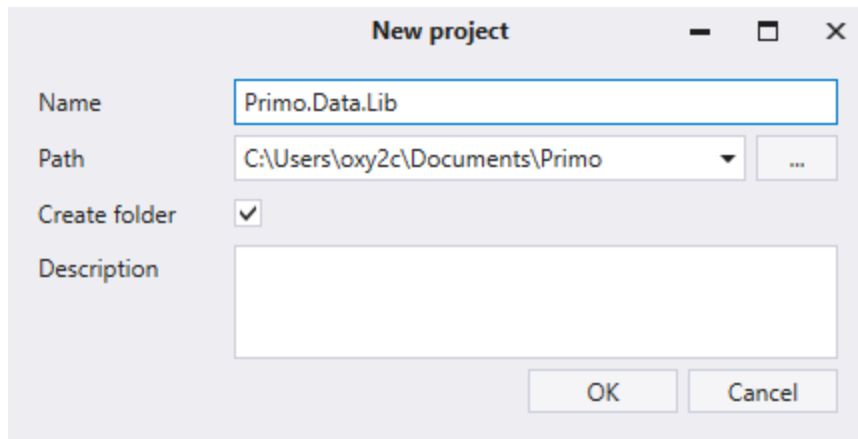
1. **Create Library** – инструмент Primo Studio, позволяющий *упаковать* `.ltw` файл в виде рабочего элемента.
2. **Primo SDK** - набор классов .NET для разработки рабочих элементов

Инструмент Create Library



Создание проекта

Чтобы создать элемент с помощью инструмента **Create Library** необходимо сначала создать проект, выполняющий нужную логику:



Создание логики

Допустим мы хотим создать простую активити для конвертации формата данных из

`Dictionary<string,string>` в `json`

Создадим для этого `.ltw` файл с соответствующим названием:

New process X

DictionaryToJson.ltw

Sequence

C#

☐ Use Orchestrator arguments ☐ Create arguments from variables

Testing

☐ Test-case





Test data path

OK Cancel

Создание логики

Создадим аргументы для нашего процесса.

- Input аргументом будет словарь `Dictionary<string,string>`
- Output аргументом будет строка `string` , содержащая словарь преобразованный в формат `json`

Arguments			
   			
	Name	Data type	Direction
▶	dictionary	System.Collections.Generic.Dictionary<Strin...	IN
	rawJson	String	OUT

Создание логики

Добавим в процесс элемент `C# Script` и напишем код, необходимый для конвертации:

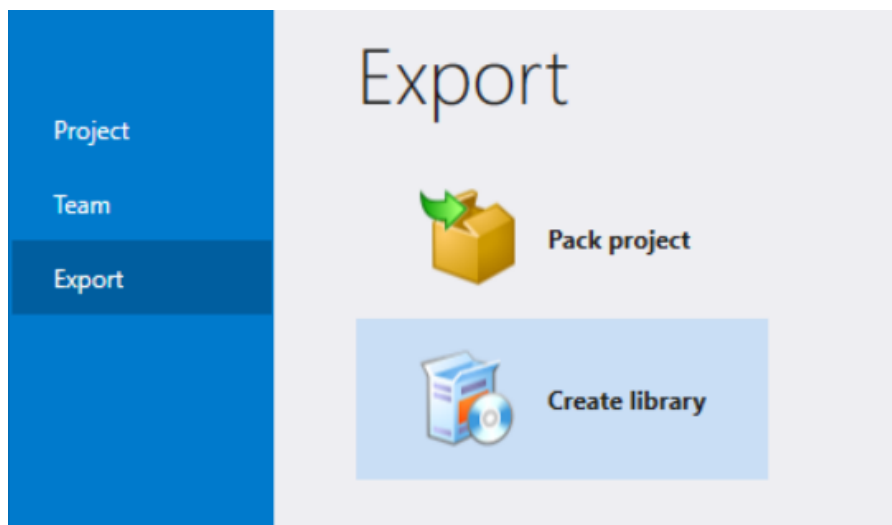
```
// Создание jobject из словаря
jobject = new Newtonsoft.Json.Linq.JObject
(
    from kvpair in dictionary
    select
        new Newtonsoft.Json.Linq.JProperty(kvpair.Key, kvpair.Value)
);

// Преобразование jobject в string
rawJson = jobject.ToString();
```

Сборка процесса

Когда процесс готов и протестирован, можно приступить к *сборке* процесса в рабочий элемент.

Для сборки перейдите в Studio в раздел: `File > Export` и выберите инструмент `Create Library` :



Откроется окно мастера упаковки ↪

Create library

Search

☒ DictionaryToJson.ltw

☐ New process.ltw

Name: Dictionary to JSON

Group: Data{\Convert}

Namespace: Primo.Data

Class: Convert

Process: DictionaryToJson.ltw

Icon: C:\Users\oxy2c\Downloads\loop-arrow.png

Help URL: https://primo-rpa.ru/

Help text: Converts Dictionary<string,string> to JSON

.NET path: C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\NETFramev

Name	Data type	Category	Property
dictionary	System.Collect...	Input	Input Dictionary
rawJson	String	Output	s

Build Cancel

Панель параметров сборки

- **Name** – название рабочего элемента
- **Group** – определяет группу в панели элементов, в которой будет находиться элемент
- **Namespace** – пространство имен, в котором будет создан класс элемента
- **Class** – имя класса, который будет содержать логику элемента
- **Process** – имя .ltw файла, который будет выполняться при вызове элемента
- **Icon** – путь до изображения – иконки элемента
- **Help URL / Help Text** – ссылка на страницу помощи, текст помощи
- **.NET Path** – путь до библиотеки .NET

Панель выбора файлов

Здесь можно выбрать .ltw файлы, которые будут включены в сборку

Панель свойств элемента

Здесь можно выбрать какие аргументы .ltw файла появятся в панели свойств. Свойствам можно задать публичное Имя (колонка Property), а также разбить по категориям (колонка Category)

Параметры сборки

Укажем параметры для сборки нашего элемента:

Название	Значение	Описание
<i>Name</i>	Dictionary to JSON	Название рабочего элемента
<i>Group</i>	Data{\}Convert	Группа в панели элементов, в которой будет находиться элемент
<i>Namespace</i>	Dictionary to JSON	Пространство имен, в котором будет создан класс элемента
<i>Class</i>	Convert	Имя класса, который будет содержать логику элемента
<i>Process</i>	DictionaryToJson.ltw	Имя <code>.ltw</code> файла, который будет выполняться при вызове элемента
<i>.NET Path</i>	*путь до .NET 4.6.2*	Путь до библиотеки .NET

Свойства элемента

Укажем свойства будущего элемента:

Name	DataType	Category	Property
dictionary	Dictionary<string,string>	Input	Input Dictionary
rawJson	string	Output	Raw JSON

Завершение сборки

После заполнения параметров сборки и свойств будущего элемента в окне мастера `Create Library` нажмём кнопку `Build`, чтобы начать сборку.

Мастер сборки предложит выбрать каталог куда будет сохранен рабочий элемент упакованный в виде `.d11` файла.

Чтобы воспользоваться созданным элементом, нужно переместить сгенерированный `.d11` файл в директорию, где установлена Primo Studio.

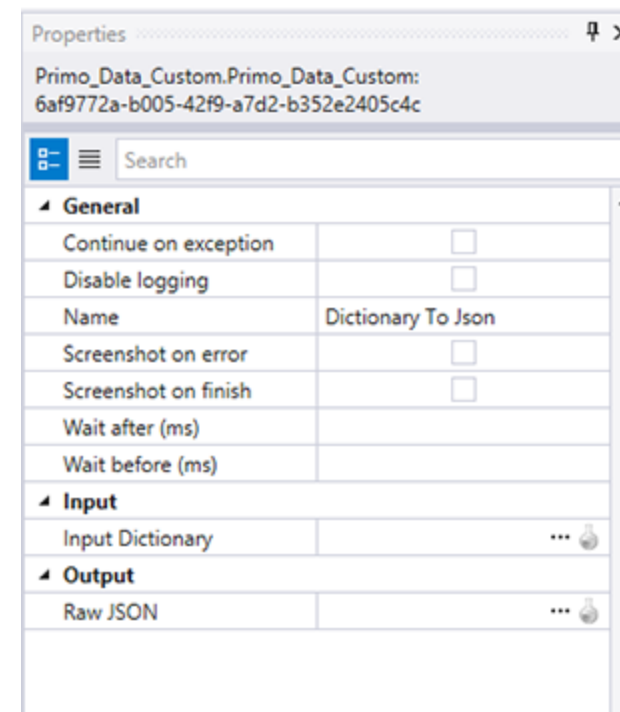
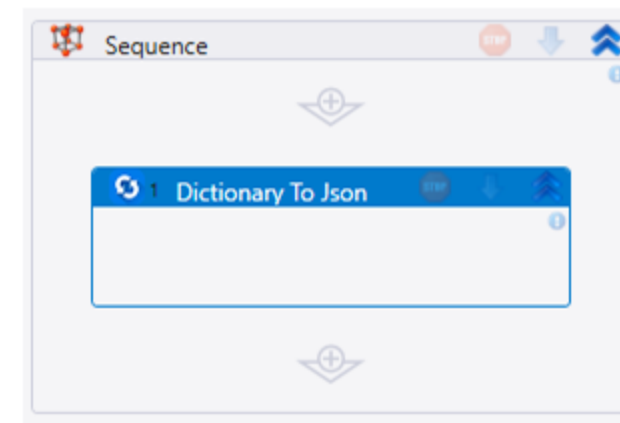
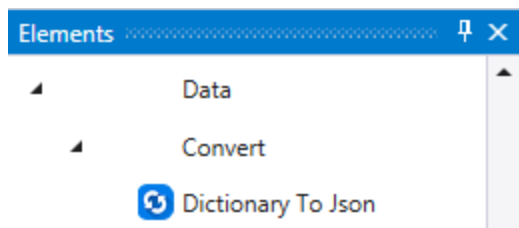
Для `x64` версии это путь:

```
C:\Program Files\Primo\Primo Studio x64
```

Проверка элемента

Теперь можно запустить Primo Studio и проверить наш элемент.

Новый элемент должен появиться в Панели Элементов в группе, которую мы указали в параметре `Group` при сборке:



Поздравляем!

Наш первый элемент готов 🎂

Primo SDK

```
1  using LTools.SDK;
2  using LTools.Common.Model;
3  using LTools.Common.UIElements;
4  ...
5  using System;
6  using System.Collections.Generic;
7  using System.Linq;
8  using System.Text;
9  using System.Threading.Tasks;
10
11 namespace Primo.NvTest
12 {
13     1 reference
14     public class ReadTextFile : PrimoComponentT0<ReadTextFileControl>
15     {
16         //Defining Elemnt Group
17         private const string CGroupName = "NV Test Group";
18         0 references
19         public override string GroupName { get => CGroupName; protected
20
21         //Time-out property
22         0 references
23         protected override int sdkTimeOut
24         {
25             get => 10000;
26             set { }
27         }
28
29         //Defining properties
30         //Prop "filePath"
31         private string prop_filePath;
32
33         [LTools.Common.Model.Serialization.StoringProperty]
34         [LTools.Common.Model.Studio.ValidateReturnScript(DataType = type
35         [System.ComponentModel.Category("File System NV")]
36         [System.ComponentModel.DisplayName("File Path display")]
37         2 references
38         public string Prop_FilePath
```

Начало работы с Primo SDK

Primo SDK – это набор классов и методов **C#** для создания элементов Primo Studio.

Для начала работы с Primo SDK понадобится IDE **Visual Studio**.

Для разработки вполне хватит версии **Community**. Это бесплатная версия, скачать ее можно [тут](#).

При установке выберите компоненты:

- .NET desktop development tools
- .NET Framework 4.6.1 targeting pack
- .NET Framework 4.6.1 SDK

Создание проекта

Откройте Visual Studio и в приветственном окне выберите `Create a new project` .
Если окно не появилось то создать новый проект можно с помощью сочетания клавиш `Ctrl + Shift + N` .

В диалоге создания проекта выберите `Class library (.NET Framework)`

Обратите внимание на язык: необходимо выбрать именно **C#**

В окне конфигурации проекта введите имя проекта.

Имя проекта должно начинаться с `Primo.` , например: `Primo.MyLibPack`

В окне конфигурации проекта выберите фреймворк: `.NET Framework 4.6.1.`

Добавление сборок

Теперь, когда проект создан необходимо добавить компоненты Primo SDK в `References` проекта. Чтобы сделать это щелкните правой кнопкой мыши по узлу `References` в панели `Solution Explorer` и выберите `Add reference...`.

В открывшемся окне `Reference Manager` нажмите кнопку `Browse...` и добавьте следующие сборки (`.dll` файлы):

- `LTools.Common.dll`
- `LTools.Dto.dll`
- `LTools.Enums.dll`
- `LTools.Scripting.dll`
- `LTools.SDK.dll`

Все эти файлы можно найти в директории, в которую установлена Primo Studio.

Добавление сборок

Помимо компонентов Primo SDK в окне `Reference Manager` также необходимо добавить следующие стандартные сборки .Net Framework:

- PresentationCore
- PresentationFramework
- System.Xaml
- WindowsBase

Из чего состоит элемент?

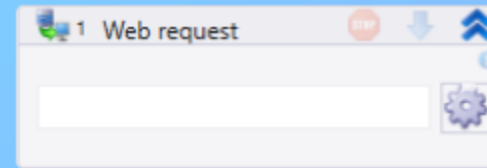
Прежде чем начинать работать над кодом будущего элемента, давайте разберемся из каких частей он должен состоять.

Любой элемент Primo состоит из:

- Графического интерфейса (**GUI**)
- Кода, выполняющего действие

Обе эти части реализованы в Primo с помощью фреймворка **Windows Presentation Foundation (WPF)**

GUI



XAML

Описывает графический интерфейс приложения

Logic

```
public override ExecutionResult TimedAction(ScriptingData sd)
{
    try
    {
        string filePath = GetPropertyValue<string>(this.Prop_filePath);
        string varName = this.Prop_fileText;
        //string varName = v;
        string fileText = System.IO.File.ReadAllText(filePath);
        //SetOutputProperty("Prop_fileText", fileText);
        SetVariableValue<string>(varName, fileText, sd);

        return new ExecutionResult() { IsSuccess = true, SuccessMessage = "File read successfully." };
    }
    catch (Exception ex)
    {
        return new ExecutionResult() { IsSuccess = false, SuccessMessage = "Error reading file: " + ex.Message };
    }
}
```

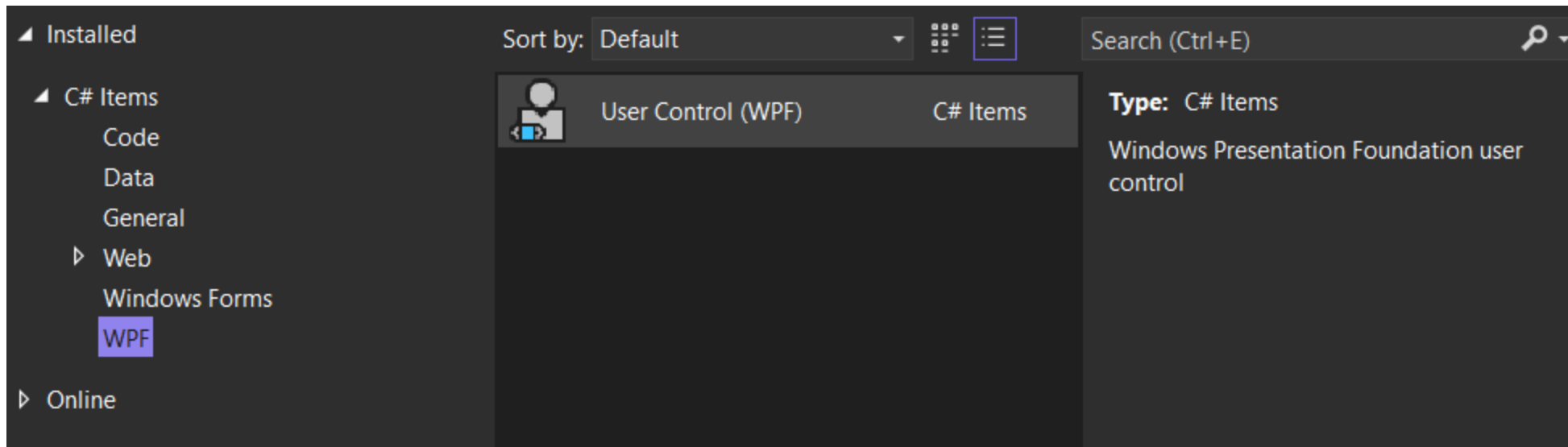
Code-Behind

Описывает функции выполняемые приложением

Добавление GUI

Теперь, когда мы понимаем из чего состоит элемент Primo, можно приступить к созданию самого элемента Primo Studio.

Для начала добавим компонент, который будет отвечать за графический интерфейс (GUI) будущего элемента. Чтобы сделать это выберите в меню Visual Studio пункт `Project > Add New Item...` (или нажмите `Ctrl + Shift + A`) и выберите компонент `User Control (WPF)`



Добавление GUI

компонент `User Control (WPF)`, который мы только что создали, представляет собой `.xaml` файл.

XAML – это язык разметки графического интерфейса в WPF, основанный на XML.

С помощью XAML мы можем добавить в наш элемент Primo Stuido UI-элементы: кнопки, поля ввода, чекбоксы, подписи и др. Полный список UI элементов можно посмотреть на сайте [Microsoft Learn](#)

Добавление GUI

Сейчас код нашей графической части выглядит так:

```
<UserControl x:Class="Primo.MyLibPack.UserControl1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:Primo.MyLibPack"
  mc:Ignorable="d"
  d:DesignHeight="450" d:DesignWidth="800">
  <Grid>
    <!-- МЕСТО ДЛЯ ВАШИХ UI-ЭЛЕМЕНТОВ -->
  </Grid>
</UserControl>
```

В атрибуте `x:Class` можно задать уникальное имя класса, описывающего GUI, например `x:Class="Primo.MyLibPack.ReadFileControl1"`

Добавление GUI

Добавим в GUI поле ввода. Чтобы сделать это поместим в тело тега `<Grid>` тег `<TextBox>` с нужными атрибутами:

```
<Grid>
  <TextBox
    Text="{Binding Prop_filePath}"
    IsReadOnly="False"
    Margin="5,0,5,0"
    Height="23"
    TextWrapping="NoWrap"
    VerticalAlignment="Center" />
</Grid>
```

Чтобы можно было получить данные, введенные в поле ввода, необходимо выполнить привязку данных (Binding) в атрибуте `Text`.

Например, чтобы связать поле ввода со свойством `PropName`, задайте атрибут `Text` в ВИДЕ: `Text="{Binding PropName}"`, где `PropName` – ИМЯ СВОЙСТВА.

Добавление code-behind

Теперь, когда графическая часть готова, можно приступить к созданию логики элемента (code-behind). Для добавления code-behind выберите в меню Visual Studio пункт `Project > Add New Item...` и выберите компонент `Class`.

Visual Studio создаст `.cs` файл следующего вида:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Primo.MyLibPack
{
    public class ReadTextFile // изменили дефолтное имя 'Class1' на 'ReadTextFile'
    {
    }
}
```

Добавление code-behind

Чтобы связать наш code-behind класс с GUI необходимо унаследовать один из двух классов Primo SDK:

- `PrimoComponentSimple` – класс для создания синхронного элемента
- `PrimoComponentT0` – класс для создания элемента с тайм-аутом

Создадим элемент с таймаутом, наследуя класс `PrimoComponentT0` :

```
public class ReadTextFile : PrimoComponentT0<UserControl1>
```

где `UserControl1` - имя класса, которое мы задавали в атрибуте `x:Class` в GUI-части

Реализация code-behind

Для функциональности элемента необходимо реализовать обязательные методы и свойства унаследанного класса. Рассмотрим их по порядку.

Свойство *GroupName*

Свойство `GroupName` отвечает за наименование группы, в которой будет содержаться элемент в Панели Элементов Primo Studio.

Для создания вложенных групп ветви дерева групп нужно разделять при помощи константы `TREE_SEPARATOR`.

Например, для формирования дерева:

- Group1
 - Group2
 - MyPrimoElement

Нужно создать имя группы вида:

```
"Group1" + WFPublishedElementBase.TREE_SEPARATOR + "Group2"
```

Свойство *sdkTimeout*

Свойство `sdkTimeout` отвечает за время ожидания завершения работы метода `TimedAction`. Если метод не завершится вовремя, будет сгенерировано исключение о тайм-ауте.

Конструктор code-behind класса

Конструктор задает свойства элемента. Их можно разделить на две группы:

- **Пользовательские свойства** – свойства, которые увидит пользователь в панели Properties в Primo Studio
- **Расширенные свойства.** К ним относятся такие свойства как:
 - `sdkComponentName` – имя элемента. Имя элемента отображается в панели Elements в Primo Studio
 - `sdkComponentHelp` – текст помощи. Помощь отображается в нижней части панели Properties в Primo Studio
 - `sdkComponentIcon` – иконка элемента. Путь к иконке выполняется в стандартном для WPF-приложений формате.

Метод *InitClass*

`InitClass` – метод, обязательный для вызова в теле конструктора code-behind класса.

`InitClass` производит сервисные действия при инициализации элемента.

Данный метод должен вызываться в конце конструктора, но перед инициализацией кастомных свойств элемента.

Принимает аргумент `container` :

```
InitClass(container);
```

Метод *SimpleAction*

`SimpleAction` – главный метод синхронного элемента.

Робот вызывает этот метод во время выполнения синхронного элемента.

Данный метод должен содержать основную бизнес-логику.

Метод принимает аргумент `sd` класса `ScriptingData`, получаемый из контекста.

Метод возвращает объект класса `ExecutionResult`.

Метод *TimedAction*

`SimpleAction` – главный метод элемента с тайм-аутом.

Робот вызывает этот метод во время выполнения элемента с тайм-аутом.

Данный метод должен содержать основную бизнес-логику.

Метод принимает аргумент `sd` класса `ScriptingData`, получаемый из контекста.

Метод возвращает объект класса `ExecutionResult`.

Класс *ExecutionResult*

Класс `ExecutionResult` служит для оповещения робота о результате выполнении элемента, а также для формирования записей в логах.

Например, чтобы оповестить об успешном завершении работы элемента, нужно создать экземпляр класса `ExecutionResult` следующего вида:

```
new ExecutionResult() { IsSuccess = true, SuccessMessage = "Completed successfully" }
```

а, чтобы сообщить об ошибке работы элемента:

```
new ExecutionResult() { IsSuccess = false, ErrorMessage = "Execution error" }
```

Схематический вид элемента с тайм-аутом

```
public class ReadTextFile : PrimoComponentT0<UserControl1>
{
    private const string CGroupName = "My Group"; // Наименование группы

    public override string GroupName
    { get => CGroupName; protected set { } }

    protected override int sdkTimeOut // Задание тайм-аута элемента в ms
    { get => 10000; set { } }

    public ReadTextFile(IWFContainer container) : base(container) // Конструктор класса
    {
        InitClass(container);
    }

    public override ExecutionResult TimedAction(ScriptingData sd) // Главный метод
    {
        return new ExecutionResult(); // Возврат результата выполнения
    }
}
```

Создание свойств

Создадим в теле code-behind класса пользовательское свойство `Prop_filePath` для хранения пути до файла:

```
private string prop_filePath; // приватное поле для хранения значения

// Атрибуты свойства
[LTools.Common.Model.Serialization.StoringProperty] // тип свойства
[LTools.Common.Model.Studio.ValidateReturnScript(DataType = typeof(string))] // тип данных
[System.ComponentModel.Category("Input")] // категория свойства в панели Properties
[System.ComponentModel.DisplayName("File Path")] // имя отображаемое в панели Properties

// Определение методов доступа к свойству (get, set)
public string Prop_filePath
{
    get { return this.prop_filePath; }
    set { this.prop_filePath = value; this.InvokePropertyChanged(this, "Prop_filePath"); }
}
```

Создание свойств

Чтобы отразить свойство `Prop_filePath` в панели Properties в Primo Studio, необходимо также добавить это свойство в конструктор code-behind класса. Для этого нужно поместить список всех свойств в Расширенное свойство `sdkProperties`:

```
sdkProperties = new List<LTools.Common.Helpers.WFHelper.PropertiesItem>()
{
    new LTools.Common.Helpers.WFHelper.PropertiesItem()
    {
        PropName = "Prop_filePath", // имя свойства
        PropertyType = LTools.Common.Helpers.WFHelper.PropertiesItem.PropertyType.SCRIPT,
        EditorType = ScriptEditorTypes.FILE_SELECTOR, // тип свойства в панели Properties
        DataType = typeof(string), // тип данных
        ToolTip = "Please specify path to a file", // всплывающая подсказка
        IsReadOnly = false
    }
}
```

Чтение свойств

Чтобы получить значение из свойства в главном методе `TimedAction`, используется метод `GetPropertyValue`.

Например, чтобы получить путь до файла из нашего свойства `Prop_filePath` в переменную `filePath`:

```
string filePath = GetPropertyValue<string>(this.Prop_filePath, "Prop_filePath", sd);
```

Запись в свойства

Допустим мы прочитали файл и хотим вернуть его содержимое в переменную `MyTextStr` типа `string`, объявленную в Primo Studio.

Для этого мы создали новое свойство `Prop_fileText` для хранения результата. Это свойство отображается в панели Properties в Primo Studio.

В панели Properties в свойстве `Prop_fileText` мы указали переменную `MyTextStr`.

Сначала элемент должен узнать имя переменной:

```
string varName = this.Prop_fileText;
```

Когда мы узнали имя переменной (`MyTextStr`), можно записать в нее результат с помощью метода `SetVariableValue`:

```
SetVariableValue<string>(varName, fileText, sd);
```


Сборка проекта

Когда вся логика элемента готова, можно собрать элемент и протестировать его в Primo Studio.

Для сборки проекта нужно выбрать конфигурацию сборки. Для этого выберите в меню Visual Studio пункт `Build > Configuration Manager...` и в выпадающем списке `Active solution configuration:` выберите `Release`

После этого можно запустить сборку, выбрав пункт меню `Build > Build Solution`

Если в проекте не было ошибок, то в подпапке `obj\Release` проекта появится файл: `Primo.[YOUR_PROJECT_NAME].dll`.

Отладка проекта

Поместите полученный после сборки `.dll` файл в директорию установки Primo Studio.

Запустите Primo Studio. Если проект не содержит ошибок, то после загрузки Primo Studio элемент станет доступен в панели Elements.

Для отладки элемента переведите отладчик Primo Studio в режим Sequence. Чтобы сделать это перейдите в Primo Studio в настройки:

`File > Settings > General > Debugger > Debugger type` и выберите режим `Sequence`.

Отладка проекта

Создайте проект и добавьте в алгоритм последовательности ваш элемент.

В Visual Studio выберите в меню пункт `Debug > Attach to Process...`, в списке процессов выберите `Primo.Studio.exe` и нажмите `Attach`.

Установите Breakpoint внутри главного метода (`SimpleAction` либо `TimedAction`).

Запустите отладку процесса в Primo Studio. Как только алгоритм дойдет до вашего элемента, Breakpoint в Visual Studio остановит выполнение и вы сможете отладить логику.

Удачи в создании собственных элементов!

Полезные ссылки:

- Документация по [инструменту Create library](#)
- Документация [Primo SDK](#)
- Список UI-элементов WPF на [Microsoft Learn](#)
- [Скачать](#) Visual Studio Community