Работа с файлами в Silverlight

При разработке сайтов, иногда возникает необходимость предоставить пользователю интерфейс для загрузки файлов, а также для управления уже загруженными файлами. Платформа Silverlight позволяет принимать файлы от клиента и передавать их серверу, в том числе в режиме Drag'n'Drop. Однако реализация работы с файлами в Silverlight будет несколько отличаться от того, что обычно делается в приложениях ASP.NET.
В этой статье рассмотрен процесс создания визуального менеджера файлов в Silverlight 4.0 на базе элемента ListBox с использованием наследования. Описан метод выполнения асинхронных HTTP-запросов при помощи классов HttpWebRequest и HttpWebResponse, а также базовые операции работы с файлами и папками.

Введение

Платформа **Silverlight** работает на стороне клиента и не может получить прямой доступ к файлам сервера. Аналогично, из соображений безопасности, имеются ограничения доступа к компьютеру пользователя. **Silverlight** не позволит передать какие-либо данные, файлы или выполнить иные действия на компьютере пользователя без его ведома и согласия.

Принципиально процесс взаимодействия приложения **Silverlight** с сервером будет такой же, как, например, в **Adobe Flash** или **JavaScript**. На сайте должна быть расположена специальная страница (шлюз), которая будет являться связующим звеном между приложением **Silverlight** и сервером. Эта страница должна уметь принимать команды от клиента, обрабатывать их, в нашем случае: иметь возможность сохранять, добавлять, изменять и удалять файлы и папки, расположенные на сервере, и в ответ отправлять результат выполнения полученной команды. Для обмена данными, между сервером и клиентом, удобней всего использовать **JSON**.

JSON (**JavaScript Object Notation**) – это текстовой формат обмена данными **JavaScript**, который главным образом ориентирован на объекты. В **.NET Framework** есть встроенные средства для быстрого преобразования (сериализации) объектов в **JSON** и обратно (десериализации). Собственно, ничто не запрещает использовать **XML**, или обычный текст. Но размер данных **XML** будет более громоздким, чем **JSON**. А текстовые данные имеет смысл использовать лишь для передачи односложных значений (например: **OK**, **ERROR** и т.п.).

Silverlight позволяет отправлять запросы удаленному серверу любыми доступными средствами **.NET Framework**, но в этой статье мы будем использовать классы **HttpWebRequest** и **HttpWebResponse**, т.к. они дают более гибкие возможности для формирования запросов. Все запросы будут делаться асинхронными методами, поскольку обычные методы могут привести к подвисанию приложения и даже всего браузера.

Как вы наверное уже догадались, наш проект будет состоять из двух основных частей: сервер и приложение **Silverlight** (клиент). Серверная часть самая простая, для универсальности она будет реализована в виде отдельного класса, который можно будет одинаково использовать как в проектах **ASP.NET WebForms**, так и **ASP.NET MVC**. В свою очередь, в приложении **Silverlight** будет сделано два независимых элемента: список файлов (менеджер файлов) и вспомогательный класс для выполнения асинхронных HTTP-запросов.

Список файлов будет реализован в виде элемента управления на основе **ListBox**, и в будущем его можно будет без проблем использовать в других проектах.
Вспомогательный класс для выполнения асинхронных запросов нам потребуется для отправки команд серверу, коих у нас будет шесть штук, чтобы не писать по шесть раз одинаковый код. Аналогично списку файлов, этот класс в будущем можно будет вынести за пределы проекта для выполнения иных задач.

Сервер

Серверная часть файлового менеджера будет работать на сервере и отвечать за обработку команд приложения **Silverlight** (клиента). Это важная часть, поскольку без неё приложение **Silverlight** не сможет получить доступ к файлам сервера. Как я уже говорил, серверная часть будет выполнена в виде отдельного класса.

Класс должен выполнять шесть основных операций: вывод списка папок/файлов, создание новых каталогов, проверка уникальности имен файлов (будет использовать перед загрузкой файла, чтобы зря не загружать сервер и сеть клиента), прием и сохранение файлов, удаление файлов/папок. Как видите, ничего сверхъестественного делать не придется, задача довольно типовая.

Итак, давайте создадим в веб-проекте (**ASP.NET WebForms** или **MVC**) класс с именем **Gateway**.

```
public class Gateway
{
}
```

Поскольку функционал класса простой, можно ограничиться всего одной основной функцией, которая будет обрабатывать клиентские запросы. Назовем её **`GetResult`**.

```
public string GetResult()
{
}
```

Функция будет возвращать строку (**string**), содержащую результат обработки запроса в формате **JSON**. В **.NET Framework** существуют встроенные средства для работы с **JSON**, находятся они в пространстве имен **System.Web.Script.Serialization** (сборка **System.Web.Extension.dll**, по умолчанию подключена). Нам потребуется лишь преобразование ответа сервера в формат **JSON**, а клиентские запросы будут передаваться в параметрах формы, как есть. Поскольку нам придется, как минимум шесть раз, проводить процесс преобразования ответа сервера в **JSON** (сериализацию), то имеет смысл сделать для этого вспомогательную функцию.

```
private StringBuilder GetJsonString(object source)
{
  StringBuilder result = new StringBuilder();
  JavaScriptSerializer myJSON = new JavaScriptSerializer();
  myJSON.Serialize(source, result);
  return result;
}
```

Для формирования ответ сервера мы будем использовать анонимные типы (городить лишние классы в данном случае смысла нет, но если есть желание, я не против). Результат выполнения

запроса клиента будет содержаться в свойстве **stat**, которое может иметь значении «**ok**», если запрос выполнен успешно, а во всех остальных случаях – «**err**». Помимо этого, в случае возникновения ошибок, сервер будет возвращать описание ошибки в свойстве **msg**. Для формирования сообщений об ошибках можно сделать еще одну вспомогательную функцию, т.к. выводить такие сообщения придется на порядок чаще, но вовсе не потому, что мы будем писать код с ошибками, напротив, мы будем перехватывать и предупреждать потенциальные ошибки.

```
private StringBuilder GetError(string msg)
{
  return GetJsonString(new { stat = "err", msg = msg });
}
```

При запросе списка файлов, помимо всего прочего, в ответе сервера нужно будет передавать два дополнительных свойства: **data** – массив файлов, **allowUp** – указатель на возможность подняться на каталог выше. Для остальных типов запросов, структура ответа сервера ничем выделяться не будет.

Примечание. *Имя **data** выбрано для универсальности, на случай, если потребуется передавать какие-либо другие данные, помимо списка файлов. В рамках данной статьи, свойство **data** кроме как для передачи списка файлов, ни для чего больше использоваться не будет.*

Но вернемся к функции **GetResult**. Мы будем обрабатывать только **POST**-запросы, это необходимо для повышения безопасности и исключения попадания страницы в поисковый индекс. В плане безопасности, конечно защита условная, но **POST**-запрос выполнить сложнее, чем **GET**, так что от хакеров-младенцев это в кой-то мере спасет. В связи с этим, параметры запроса нам нужно будет искать среди переменных формы (**Request.Form**). Получить доступ к объекту **Request** в классе можно через **HttpContext.Current**. Чтобы постоянно не писать длинный путь к классу **Request**, в самом начале функции объявим вспомогательные переменные, со ссылками на нужные классы контекста (**HttpContext**).

```
HttpRequest Request = HttpContext.Current.Request;
HttpServerUtility Server = HttpContext.Current.Server;
```

Примечание. *Классы **HttpContext**, **HttpRequest**, **HttpServerUtility** принадлежат пространству имен **System.Web** и находятся в сборке **System.Web.dll** (по умолчанию в веб-проектах дополнительно подключать не нужно, но если вы будете делать класс в виде отдельной библиотеки, эта информация может быть полезна).*

Для формирования результата обработки команды, мы будем использовать переменную **result** типа **StringBulder**.

```
StringBuilder result = new StringBuilder();
```

Примечание. *Класс **StringBuilder** находится в пространстве имен **System.Text**.*

Основные параметры запроса, для удобства, лучше скопировать в отдельные переменные. Основных параметров у нас будет два: **cmd** – команда, которую нужно выполнить; **path** – путь, в котором нужно выполнить команду.

```
string cmd = "", path = "";
if (!String.IsNullOrEmpty(Request.Form["cmd"])) { cmd = Request.Form["cmd"].ToLower(); }
if (!String.IsNullOrEmpty(Request.Form["path"])) { path = Request.Form["path"]; } else
{ path = "/"; }
if (!path.EndsWith("/")) path += "/";
```

Путь (**path**) будет формироваться относительно корневого каталога сайта. Но для большей гибкости, можно объявить еще одну переменную, которая будет содержать верхний каталог, в котором клиенты смогут работать с файлами.

```
string _Root = ""; // имя корневого каталога, если пусто – корень сайта
```

Примечание. *Клиентам лучше никогда не предоставлять доступ к корневому каталогу сайта.*

Далее, нужно проверить, чтобы запрашиваемый клиентом каталог существовал. Если каталог не существует, то сервер должен возвращать сообщение об ошибке.

```
DirectoryInfo DI = new DirectoryInfo(Server.MapPath(String.Format("~/{0}{1}", _Root,
path)));
if (!DI.Exists)
{
  result = GetError(String.Format("Ошибка. Каталог \"{0}\" не найден.", String.Format("~/
{0}{1}", _Root, path)));
  return result.ToString();
}
```

Примечание. *Для использования класса **DirectoryInfo** может потребоваться импортировать в проект пространство имен **System.IO**.*

Если каталог существует, то можно приступить к обработке команды (**cmd**) клиента. Процесс обработки команды будет представлен в виде условия, первым элементом которого будет выполнение команды на проверку уникальности имени файла/папки (**check**). Как я уже говорил ранее, эту команду будет полезно использовать перед отправкой файла на сервере. Поскольку размер файла может быть существенным, то лучше заранее проверить, сможет ли сервер его сохранить или нет, чтобы зяря не нагружать сервер и не расходовать трафик клиента. Я буду проверять только уникальность имени файла, но вообще здесь можно реализовать проверку типа (расширения), размера файла и т.п., или напротив ничего не проверять и разрешать все. Сервер ожидает имя файла в переменной формы **name**.

```
if (cmd == "check")
{
  #region проверка уникальности имени файла/папки
  if (File.Exists(Path.Combine(DI.FullName, Request.Form["name"])))
  {
    result = GetJsonString(new { stat = "err", msg = String.Format("Файл с именем \"{0}\"
уже есть в каталоге \"{1}\".", Request.Form["name"], path) });
  }
  else
  {
    result = GetJsonString(new { stat = "ok" });
  }
  #endregion
}
```

Второй операцией (**cmd**) будет, собственно, сохранение полученного файла на сервере (**upload**). Файл ожидается в параметре **file1**. Перед загрузкой, по-хорошему, необходимо выполнить все действия, которые описаны в обработчике команды **check**. Я дополнительный код проверки писать не стал, так что если файл с указанным именем существует на сервере, то он просто будет перезаписан.

Примечание. *Имя параметра, в котором передается файл –* ***file1*** *может быть другим, и более того, в одном запросе можно передавать несколько файлов. В этом нам помогут классы* ***HttpWebRequest*** *и* ***HttpWebResponse****, о которых пойдет речь в следующей части статьи.*

```
else if (cmd == "upload")
{
  #region загрузка файла
  if (Request.Files["file1"] == null || Request.Files["file1"].ContentLength <= 0)
  {
    result = GetError("Ошибка. Файл не найден.");
  }
  else
  {
    using (FileStream fs = System.IO.File.Create(Path.Combine(DI.FullName,
Request.Files["file1"].FileName)))
    {
      byte[] buffer = new byte[4096];
      int bytesRead;
      while ((bytesRead = Request.Files["file1"].InputStream.Read(buffer, 0,
buffer.Length)) != 0)
      {
        fs.Write(buffer, 0, bytesRead);
      }
    }
    result = GetJsonString(new { stat = "ok" });
  }
  #endregion
}
```

Следующей командой будет обработка запроса на создание нового каталога (**newdir**). Имя каталога ожидается в переменной формы с именем «**name»**.

```
else if (cmd == "newdir")
{
  #region создание новой папки
  if (String.IsNullOrEmpty(Request.Form["name"]))
  {
    result = GetError("Ошибка. Отсутствует имя папки.");
  }
  else
  {
    DirectoryInfo d = new DirectoryInfo(Path.Combine(DI.FullName, Request.Form["name"]));
    if (d.Exists)
    {
      result = GetError("Ошибка. Папка с таким именем уже существует.");
    }
    else
    {
      d.Create();
      result = GetJsonString(new { stat = "ok" });
    }
  }
  #endregion
```

```
}
```

Четвертым элементом условия будет обработка запроса на удаление файла или папки (**delete**). Здесь придется проверять, чем является удаляемый объект, файлом или каталогом. Удаление каталога будет происходить рекурсивно, т.е. сначала из папки должны быть удалены все вложенные объекты, а уже после, сам каталог, в противном случае произойдет ошибка.
Имя удаляемого объекта ожидается в поле формы с именем «**name»**.

```
else if (cmd == "delete")
{
  #region удаление файла/папки
  if (String.IsNullOrEmpty(Request.Form["name"]))
  {
    result = GetError("Ошибка. Отсутствует имя файла/папки.");
  }
  else
  {
    if (File.GetAttributes(Path.Combine(DI.FullName, Request.Form["name"])) ==
FileAttributes.Directory)
    {
      Directory.Delete(Path.Combine(DI.FullName, Request.Form["name"]), true);
    }
    else
    {
      File.Delete(Path.Combine(DI.FullName, Request.Form["name"]));
    }
    result = GetJsonString(new { stat = "ok" });
  }
  #endregion
}
```

Еще одна команда, которую будет выполнять сервер – это изменение имени файла/папки (**rename**). Здесь, по аналогии с удалением, необходимо определить тип объекта и переименовать при помощи класса **Directory** или **File**, а точнее переместить (**Move**). При этом сервер должен получить от клиента как старое (текущее) имя (**oldName**), так и новое имя (**newName**) объекта.

```
#region изменение имени файла/папки
string oldName = Request.Form["oldName"], newName = Request.Form["newName"];
if (String.IsNullOrEmpty(oldName) || String.IsNullOrEmpty(newName))
{
  result = GetError("Ошибка. Отсутствует имя файла/папки.");
}
else
{
  if (newName != oldName)
  {
    if (File.GetAttributes(Path.Combine(DI.FullName, oldName)) ==
FileAttributes.Directory)
    {
      Directory.Move(Path.Combine(DI.FullName, oldName), Path.Combine(DI.FullName,
newName));
    }
    else
    {
      File.Move(Path.Combine(DI.FullName, oldName), Path.Combine(DI.FullName, newName));
    }
  }
  result  = GetJsonString(new { stat = "ok" });
}
#endregion
```

Последним, шестым, элементом условия будет обработка запроса на получение списка файлов и папок. Этот элемент условия будет выполняться по умолчанию для всех неизвестных команд (**cmd**). Список файлов и папок будет передан в свойство **data**. Каждый элемент списка в обязательном порядке будет содержать имя файла/папки (**name**), размер (**size**), тип (0 - папка, 1 - файл) и **url**.

```
else
{
  #region список файлов (по умолчанию)
  ArrayList files = new ArrayList();
  foreach (DirectoryInfo d in DI.GetDirectories())
  {
    files.Add(new
    {
      name = d.Name,
      size = 0,
      type = 0,
      url = String.Format("http://{0}/{1}{2}{3}", Request.Url.Host + (Request.Url.Port >
80 ? ":" + Request.Url.Port.ToString() : ""), _Root, path, d.Name)
    });
  }
  foreach (FileInfo f in DI.GetFiles())
  {
    files.Add(new
    {
      name = f.Name,
      size = f.Length,
      type = 1,
      url = String.Format("http://{0}/{1}{2}{3}", Request.Url.Host + (Request.Url.Port >
80 ? ":" + Request.Url.Port.ToString() : ""), _Root, path, f.Name)
    });
  }
  bool allowUp = !String.IsNullOrEmpty(path.Trim("/".ToCharArray()));
  result = GetJsonString(new { stat = "ok", allowUp = allowUp, data = files });
  #endregion
}
```

Вот и все, класс готов. Для исключения неожиданных ошибок, лучше поместить содержимое функции **GetResult** в блоки **try {} catch {}**.

```
public string GetResult()
{
  if (HttpContext.Current == null) throw new Exception("Ожидается HTTP-запрос.");

  HttpRequest Request = HttpContext.Current.Request;
  HttpServerUtility Server = HttpContext.Current.Server;

  StringBuilder result = new StringBuilder();

  try
  {
    string cmd = "", path = "";
    //... весь код, указанный по тексту выше
  catch (Exception ex)
  {
    result = GetError(ex.Message);
  }
  return result.ToString();
}
```

Теперь можно использовать этот класс при создании страницы, которая будет обрабатывать запросы от приложения **Silverlight**. В проектах **ASP .NET WebForms** для этого лучше всего сделать универсальный обработчик (**.ashx** - хендлер, **ASP.NET Handler**), например **FileManagerGateway.ashx**.

```
public class FileManagerGateway : IHttpHandler
{

  public void ProcessRequest(HttpContext context)
  {
    Gateway myGateway = new Gateway();
    // указываем тип содержимого - JSON
    context.Response.ContentType = "application/json";
    // выводим результат обработки запроса
    context.Response.Write(myGateway.GetResult());
  }

  public bool IsReusable
  {
    get
    {
      return false;
    }
  }
}
```

В проектах **ASP .NET MVC** достаточно прописать действие (**Action**) в контроллере (**Controller**). Например, в контроллере **HomeController** действие **FileManagerGateway** (создавать **View** необязательно).

```
[HttpPost]
public ActionResult FileManagerGateway()
{
  Gateway myGateway = new Gateway();
  return new ContentResult() { Content = myGateway.GetResult(), ContentType =
"application/json", ContentEncoding = System.Text.Encoding.UTF8 };
}
```

С сервером мы разобрались, теперь можно приступать непосредственно к созданию приложения **Silverlight**.

Приложение Silverlight

Приложение **Silverlight** будет состоять из двух независимых частей. Первая часть – это список файлов. Вторая – класс для выполнения асинхронных запросов. Начнем, пожалуй, с конца.

Вспомогательный класс для выполнения асинхронных запросов

Нам придется отправлять на сервер как минимум шесть команд и, причем, делать это нужно будет асинхронными методами. Использование любых асинхронных методов выглядит немного сложнее, чем обычных. Кода выйдет заметно больше, а структура получится, на первый взгляд, менее логичная. Все запросы у нас будут типовыми, однако с некоторыми существенными различиями, которые нужно будет учитывать при создании класса. Создание отдельного класса

позволит нам значительно сократить объем кода, а также упростит процесс его отладки и улучшения. Класс назовем **WebHelper**.

```
public class WebHelper
{
}
```

Наш класс должен будет отправлять запросы на сервер, для этого ему необходимо знать, куда их отправлять. Добавим свойство, которое будет содержать адрес странички (**Url**).

```
private string _Url = String.Empty;
public string Url
{
  get { return _Url; }
  set { _Url = value; }
}
```

Запросы у нас будут отправляться только методом **POST**, но нам ничто не мешает реализовать в классе и поддержку метода **GET** (в рамках данной статьи метод **GET** использоваться не будет).

```
private string _Method = "POST";
public string Method
{
  get { return _Method; }
  set
  {
    _Method = value;
    if (String.IsNullOrEmpty(_Method) || _Method.ToUpper() != "GET" ||
_Method.ToUpper() != "POST") _Method = "POST";
  }
}
```

Добавленные свойства (**url** и **method**) являются обязательными, поэтому имеет смысл принимать их при создании экземпляра класса, т.е. описать в конструкторе.

```
public WebHelper(string url) : this (url, "POST") { }
public WebHelper(string url, string method)
{
  _Url = url;
  _Method = method;
}
```

Каждый запрос у нас будет иметь параметры, причем параметры могут быть как текстовыми, так и содержать файлы. Сделаем еще один класс, который будет представлять параметр запроса. Назовем его **QueryItem**.

```
public class QueryItem
{
}
```

Параметр должен иметь имя и значение, сделаем для этого два свойства – **Name** и **Value**. Чтобы иметь возможность записывать в качестве значения параметра текст и файл, нужно использовать объектный тип данных (**object**).

```
public string Name { get; set; }
public object Value{get;set;}
```

При передаче файлов на сервер нужно будет указывать еще и имя файла, для этого мы сделаем свойство **FileName**.

```
public string FileName { get; set; }
```

Помимо этого, для файлов нужно будет передавать тип содержимого (**Content-Type**) для этого мы добавим функцию **GetContentType**. Определять тип содержимого можно по расширению файла. В **Silverlight**, поскольку доступ к реестру ограничен, для этого можно создать коллекцию сопоставления типов содержимого с расширениями. В приведенном ниже фрагменте кода эта функция всегда будет возвращать «**application/data**», однако в файле проекта, прикрепленного к статье, я реализовал функцию в полной мере.

```
public string GetContentType()
{
  return "application/data";
}
```

Запросы, с файлами и без, будут отличаться и нам потребуется определять, есть ли среди параметров файлы или нет. Чтобы упростить этот процесс, сделаем дополнительное свойство – **IsFile**, которое будет проверять тип **Value**.

```
public bool IsFile
{
  get
  {
    return this.Value != null && this.Value.GetType() == typeof(FileStream);
  }
}
```

Для каждого типа параметра опишем свой конструктор.

```
public QueryItem(string name, string value)
{
  this.Name = name;
  this.Value = value;
}

public QueryItem(string name, string fileName, Stream stream)
{
  this.Name = name;
  this.FileName = fileName;
  this.Value = stream;
}
```

Последним штрихом в этом классе добавим две вспомогательные функции.
Функция **ValueForUrl** будет проводить кодирование значения (**Value**) для использования в **Url**. Эта функция пригодится при формировании простых **POST** и **GET** запросов.

```
public string ValueForUrl()
{
  return HttpUtility.UrlEncode(this.Value.ToString());
}
```

Функция **ValueAsString** будет возвращать строку, т.к. значение имеет объектный тип данных (**object**), чтобы постоянно не писать **Value.ToString()**. Эта функция будет полезна для текстовых значений при формировании запросов, содержащих файлы.

```
public string ValueAsString()
{
  return this.Value.ToString();
}
```

Теперь мы можем создать в нашем классе **WebHelper** коллекцию параметров для формирования запроса. Для этого можно использовать коллекцию **List<QueryItem>**. Но если подумать, можно сделать свою коллекцию. Почему нет? Ну да, я конечно знаю, чем все закончится, а вам пока не видно. Сделаем класс **QueryItemCollection**, который будет наследоваться от класса **List<QueryItem>**.

```
public class QueryItemCollection : List<QueryItem>
{
}
```

В классе **QueryItem** у нас два конструктора и теперь мы можем в нашей коллекции сделать два вспомогательных метода для добавления новых элементов (параметров) в коллекцию.

```
public void Add(string name, string value)
{
  this.Add(new QueryItem(name, value));
}
public void Add(string name, string fileName, Stream stream)
{
  this.Add(new QueryItem(name, fileName, stream));
}
```

В будущем, для **GET**-запросов и простых **POST**-запросов (т.е. как минимум два раза) нам нужно будет получать параметры в виде одной строки (*par1=val1&par2=val2*), где значения должны быть кодированы для **Url**. Сделаем для этого отдельную функцию.

```
public string GetQueryString()
{
  string qs = "";
  foreach (QueryItem itm in this)
  {
    if (!String.IsNullOrEmpty(qs)) qs += "&";
    qs += String.Format("{0}={1}", itm.Name, itm.ValueForUrl());
  }
  return qs;
}
```

Чтобы определить, какой тип содержимого нужно использовать для **POST**-запроса, нам необходимо знать, есть ли среди параметров запроса файлы или нет. Добавим функцию, которая проверит коллекцию на наличие файлов.

```
public bool HasFiles()
{
  foreach (QueryItem itm in this)
  {
    if (itm.IsFile) return true;
  }
  return false;
```

```
}
```

Коллекция готова. Вернемся к классу **WebHelper** и добавим свойство для работы с параметрами запроса.

```
private QueryItemCollection _Queries = new QueryItemCollection();
public QueryItemCollection Queries
{
  get
  {
    return _Queries;
  }
  set
  {
    _Queries = value;
  }
}
```

Для запросов с файлами, нам понадобится создавать границу (**Boundary**), которая будет разделять элементы запросы друг от друга, т.е. каждый параметр/файл будут записываться в запрос отдельно. Для этого сделаем локальную переменную на уровне класса (**WebHelper**).

```
private string _Boundary = String.Empty;
```

Теперь, когда все необходимое у нас есть, приступим к реализации выполнения HTTP-запроса. Добавим для этого публичный метод **Execute**.

```
public void Execute()
{
}
```

В методе **Execute** нужно создать экземпляр класса **HttpWebRequest** и установить ему свойства, в зависимости от типа запроса. Для запросов типа **GET**, если в запросе есть параметры, их нужно добавить к **url**.

```
// для GET-запросов добавляем параметры к Url
string url = _Url;
if (_Method == "GET")
{
  string qs = _Queries.GetQueryString();
  if (url.EndsWith("?"))
  {
    url += "&" + qs;
  }
  else
  {
    url += "?" + qs;
  }
}

// создаем экземпляр класса HttpWebRequest
HttpWebRequest myReq = (HttpWebRequest)HttpWebRequest.Create(_Url);
myReq.Method = _Method;
```

Запросы типа **POST** могут быть двух типов: с файлами и без файлов. Запросы без файлов (простые) должны иметь тип содержимого **application/x-www-form-urlencoded**. А запросы с файлами

должны иметь тип содержимого **multipart/form-data** и границу, для разделения этого содержимого. Для этого мы воспользуемся функцией **HasFiles** коллекции параметров.

```
if (_Method == "POST")
{
  if (_Queries.HasFiles())
  {
    _Boundary = "----------" + DateTime.Now.Ticks.ToString("x");
    myReq.ContentType = "multipart/form-data; boundary=" + _Boundary;
  }
  else
  {
    myReq.ContentType = "application/x-www-form-urlencoded";
  }
}
```

Примечание. *Можно отправлять все **POST**-запросы с типом содержимого **multipart/form-data**, но тогда запрос будет более сложным и будет занимать больше пространства, чем **application/x-www-form-urlencoded**.*

Чтобы приступить к добавлению параметров запроса, нужно вызвать метод **BeginGetRequestStream** экземпляра **HttpWebRequest**. Этот метод проверит, готов ли сервер принять наш запрос или нет.

```
myReq.BeginGetRequestStream(Execute_BeginGetRequestStream, myReq);
```

Если сервер готов принять запрос, управление будет переведено в процедуру - **Execute_BeginGetRequestStream**, в противном случае произойдет исключение.

```
private void Execute_BeginGetRequestStream(IAsyncResult result)
{
  HttpWebRequest r = result.AsyncState as HttpWebRequest; // получаем запрос
}
```

Если текущий запрос выполняется методом **POST** и имеет какие-то параметры, мы должны их передать серверу. Для этого будет использоваться переменная **myStream**.

Примечание. *Для запросов типа **GET** параметры передавать отдельно не нужно, т.к. они уже находятся в **Url**.*

```
if (_Queries.Count > 0 && _Method == "POST")
{
  Stream myStream = r.EndGetRequestStream(result);
  // код записи параметров в запос будет здесь
  myStream.Close();
}
```

Если в запросе нет границы (**Bounday**), записываем параметры в виде строки, полученной из функции **GetQueryString**.

```
if (String.IsNullOrEmpty(_Boundary))
{
  byte[] buffer = Encoding.UTF8.GetBytes(_Queries.GetQueryString());
  myStream.Write(buffer, 0, buffer.Length);
}
```

Если граница есть, значит запрос содержит файлы, и для каждого параметра нужно указывать отдельные заголовки и разделять содержимое границей.

```
else
{
  byte[] buffer = null;
  foreach (QueryItem itm in _Queries)
  {
    if (!itm.IsFile)
    {
      // это обычный параметр
      string q = String.Format("\r\n--{0}\r\nContent-Disposition: form-data;
name=\"{1}\";\r\n\r\n{2}", _Boundary, itm.Name, itm.ValueAsString());
      buffer = Encoding.UTF8.GetBytes(q);
      myStream.Write(buffer, 0, buffer.Length);
    }
    else
    {
      // это файл
      string q = String.Format("\r\n--{0}\r\nContent-Disposition: form-data;
name=\"{1}\"; filename=\"{2}\"\r\nContent-Type: {3}\r\n\r\n", _Boundary, itm.Name,
itm.FileName, itm.GetContentType());
      buffer = Encoding.UTF8.GetBytes(q);
      // заголовки файла
      myStream.Write(buffer, 0, buffer.Length);
      // тело файла
      buffer = new byte[4096]; // размер буфера чтения 4 Кб
      int bytesRead = 0; int totalSize = 0;
      while ((bytesRead = ((Stream)itm.Value).Read(buffer, 0, buffer.Length)) != 0) //
читаем файл
      {
        myStream.Write(buffer, 0, buffer.Length); // пишем файл в запрос
        totalSize += bytesRead;
      }
    }
  }
  // закрываем границу
  buffer = Encoding.UTF8.GetBytes(String.Format("\r\n--{0}--\r\n", _Boundary));
  myStream.Write(buffer, 0, buffer.Length);
}
```

После того, как параметры будут записаны, можно попробовать получить ответ от удаленного сервера при помощи метода **BeginGetResponse** экземпляра класса **HttpWebRequest**. Ответ сервера будет передан в нашу процедуру **Execute_Complete**.

```
r.BeginGetResponse(Execute_Complete, r);
```

В методе **Execute_Complete** можно обработать полученный от сервера ответ.

```
private void Execute_Complete(IAsyncResult result)
{
  HttpWebRequest myReq = (HttpWebRequest)result.AsyncState;
  HttpWebResponse myResp = (HttpWebResponse)myReq.EndGetResponse(result);

  if (myResp.StatusCode == HttpStatusCode.OK)
  {
    // считываем ответ сервера
    StreamReader reader = new StreamReader(myResp.GetResponseStream(), Encoding.UTF8);
    string page = reader.ReadToEnd();
    // в переменной page содержится ответ сервера
  }
  else
  {
    // ошибка сервера
  }
}
```

Класс почти готов. Использовать его довольно просто.

```
WebHelper w = new WebHelper("http://localhost:123/FileManagerGateway.ashx");
w.Queries.Add("cmd", "get");
w.Queries.Add("path", _Path);
w.Execute();
```

Однако класс должен быть достаточно универсальным, а сейчас выходит, что для обработки ответа придется писать дополнительные условия и функции. Чтобы все было совсем красиво, для реализации файлового менеджера мы будем проводить предварительную обработку ответа сервера в процедуре **Execute_Complete** и передавать результат в функцию обратного вызова. Если вы еще не забыли, сервер возвращает данные в формате **JSON**, которые могут содержать следующие поля: **stat**, **msg**, **allowUp** и **data**. Именно эти поля мы и будем передавать в функцию обратного вызова. Для этого объявим в классе **WebHelper** делегат.

```
public delegate void WebCallback(string stat, string msg, bool allowUp, JsonValue data,
object tag);
```

Примечание. *Для использования класса **JsonValue** необходимо подключить к проекту сборку **System.Json.dll** (Проект -> Добавить ссылку -> .NET -> System.Json.dll) и импортировать пространство имен **System.Json**.*

Как видите, я еще добавил параметр **tag** типа **object**. Он понадобится нам в будущем. В классе **WebHelper** параметр **tag** должен быть отражен в виде свойства **Tag**.

```
public object Tag { get; set; }
```

Также на уровне класса (**WebHelper**) нужно создать переменную, которая будет хранить ссылку на функцию обратного вызова.

```
private WebCallback _Callback = null;
```

А принимать ссылку на функцию обратного вызова у нас будет метод **Execute**.

```
public void Execute(WebCallback callback)
{
  _Callback = callback;
  // остальной код, который мы писали ранее
}
```

И теперь в коде процедуры **Execute_Complete** можно десериализовать ответ сервера и передать его в функцию обратного вызова.

```
private void Execute_Complete(IAsyncResult result)
{
  HttpWebRequest myReq = (HttpWebRequest)result.AsyncState;
  HttpWebResponse myResp = (HttpWebResponse)myReq.EndGetResponse(result);

  string stat = "", msg = "";
  bool allowUp = false;
  JsonValue data = null;

  if (myResp.StatusCode == HttpStatusCode.OK)
  {
    // считываем ответ сервера
    StreamReader reader = new StreamReader(myResp.GetResponseStream(), Encoding.UTF8);
    string page = reader.ReadToEnd();
    // парсим JSON
    JsonValue json = System.Json.JsonObject.Parse(page);
    // передаем полученные данные в переменные
    if (json.ContainsKey("stat")) stat = json["stat"];
    if (json.ContainsKey("msg")) msg = json["msg"];
    if (json.ContainsKey("allowUp")) allowUp = json["allowUp"];
    if (json.ContainsKey("data")) data = json["data"];
  }
  else
  {
    stat = "err";
    msg = String.Format("Ошибка сервера {0}", myResp.StatusCode);
  }

  if (_Callback != null)
  {
    _Callback(stat, msg, allowUp, data, this.Tag);
  }
}
```

Тогда код использования класса будет примерно таким:

```
private void Run()
{
  WebHelper w = new WebHelper("http://localhost:123/FileManagerGateway.ashx");
  w.Queries.Add("cmd", "get");
  w.Queries.Add("path", _Path);
  w.Execute(RunResult);
}

private void RunResult(stat, string msg, bool allowUp, JsonValue data, object tag)
{
  // обработка ответа сервера
}
```

Т.е. в функции обратного вызова мы с вами в итоге имеем ответ сервера на более высоком уровне, чем он был изначально (в виде переменных, а не **JSON**). Это конечно не так универсально

и я оптимизировал код для решения нашей задачи по созданию файлового менеджера. Но вы можете легко переделать механизм передачи ответа сервера в функцию обратного вызова. Для этого достаточно немного изменить делегат. Например, можно передать экземпляр класса **HttpWebResponse**.

```
public delegate void WebCallback(HttpWebResponse resp);
// ...
if (_Callback != null)
{
  _Callback(myResp);
}
```

В плане того, какие данные передавать в функцию обратного вызова, все ограничивается лишь вашей фантазией.

Класс полностью готов к использованию, теперь перейдем к созданию элемента для вывода списка файлов.

Список файлов

Для отображения списка файлов мы возьмем за основу стандартный элемент управления **ListBox**, наполнив его необходимым функционалом. Помимо обычного вывода данных, наш **ListBox** должен уметь отправлять запросы удаленному серверу и обрабатывать полученные ответы. Для чего мы, собственно, и сделали класс **WebHelper**. Данные в списке у нас буду не совсем обычные, вывести имена файлов/папок в текстовом виде конечно просто, но куда более интересней выделить их графически, а в перспективы использовать уникальные иконки для разных типов файлов. В **Silverlight** это делается заметно проще, чем например в **Windows Forms**.

Класс FileItem (элемент списка)

Поскольку элементы списка у нас будут нестандартными, разумней сделать для этого отдельный класс. Назовем его **FileItem**, класс будет наследоваться от **StackPanel**.

```
public class FileItem : StackPanel
{
}
```

Как вы помните, на запрос списка файлов, сервер возвращает массив данных в свойстве **data**. Каждый элемент массива содержит имя файла (**name**), размер (**size**), тип (0 - папка, 1 - файл) и **url**. Следовательно, наш будущий элемент списка должен иметь аналогичные свойства.

```
public int ItemType { get; set; }
public string FileName { get; set; }
public double FileSize { get; set; }
public string FileUrl { get; set; }
```

Принимать значения для этих свойств можно в конструкторе класса.

```
public FileItem(int type, string name, string url, double size)
{
  this.ItemType = type;
  this.FileName = name;
  this.FileUrl = url;
  this.FileSize = size;
}
```

И в нем же можно формировать визуальную составляющую элемента списка. Визуально, элемент списка будет состоять из иконки, имени, трех кнопок (открыть, переименовать, удалить) и индикатора размера файла. Также следует учесть, что в списке должна быть реализована возможность перехода на верхний уровень из вложенного каталога. Для этого, к уже известным типам элементов (0 - папка, 1 - файл), добавим еще один тип – минус один (-1), который и будет указывать на то, что это переход на верхний уровень.

Примечание. *Физически добавлять тип элемента никуда не надо, имеется ввиду, что необходимо не забыть учесть его при написании кода.*

Поскольку элемент списка наследуется от класса **StackPanel**, который является контейнером, то в него можно добавить любые другие элементы. Для начала необходимо переключить контейнер в режим горизонтального выравнивания дочерних элементов управления.

```
this.Orientation = Orientation.Horizontal;
```

Затем можно добавить иконку, в зависимости от типа объекта (-1 - папка верхнего уровня, 0 – обычная папка, 1 - файл).

```
// иконка
Image myImg = new Image() { Width = 16, Height = 16 };
if (type == -1)
{
  // это корневая папка, позволяющая перейти на уровень выше
  myImg.Source = new System.Windows.Media.Imaging.BitmapImage(new
Uri("Images/folder2.png", UriKind.Relative));
}
else if (type == 0)
{
  // это папка
  myImg.Source = new System.Windows.Media.Imaging.BitmapImage(new
Uri("Images/folder.png", UriKind.Relative));
}
else
{
  // это файл
  string fileExtension = System.IO.Path.GetExtension(name).ToLower();
  myImg.Source = new System.Windows.Media.Imaging.BitmapImage(new
Uri("Images/unknown.png", UriKind.Relative));
}
myImg.Margin = new Thickness(2, 0, 0, 0);

// добавляем иконку
this.Children.Add(myImg);
```

Имя файла/папки в виде нередактируемого текстового блока (**TextBlock**).

```
// добавляем имя файла/папки
this.Children.Add(new TextBlock() { Text = name, Margin = new Thickness(2, 0, 0, 0) });
```

Иконки-кнопки для управления файлами/папками. Причем кнопки изменения имени и удаления не должны быть доступны для папок, которые ведут на верхний уровень (**ItemType** = -1). Каждой кнопке нужно установить обработчик клика левой кнопкой мышки (подробнее об этом далее).

```
// иконка для открытия файла/папки
Image myImg2 = new Image() { Width = 9, Height = 9, Cursor = Cursors.Hand };
myImg2.Margin = new Thickness(4, 0, 0, 0);
myImg2.Source = new System.Windows.Media.Imaging.BitmapImage(new Uri("Images/open.png",
UriKind.Relative));
myImg2.MouseLeftButtonUp += (sender, e) =>
{
  // открываем файл/папку
  Open();
};
this.Children.Add(myImg2);

if (type != -1)
{
  // иконка для изменения имени файла/папки
  Image myImg4 = new Image() { Width = 9, Height = 9, Cursor = Cursors.Hand };
  myImg4.Margin = new Thickness(4, 0, 0, 0);
  myImg4.Source = new System.Windows.Media.Imaging.BitmapImage(new Uri("Images/edit.png",
UriKind.Relative));
  myImg4.MouseLeftButtonUp += (sender, e) =>
  {
    EditStart();
  };
  this.Children.Add(myImg4);

  // иконка для удаления файла/папки
  Image myImg3 = new Image() { Width = 9, Height = 9, Cursor = Cursors.Hand };
  myImg3.Margin = new Thickness(4, 0, 0, 0);
  myImg3.Source = new System.Windows.Media.Imaging.BitmapImage(new Uri("Images/del.png",
UriKind.Relative));
  myImg3.MouseLeftButtonUp += (sender, e) =>
  {
    Delete();
  };
  this.Children.Add(myImg3);
}
```

Примечание. *Предполагается, что в проекте есть файлы изображений **folder.png**, **folder2.png**, **unknown.png**, **open.png**, **edit.png**, **del.png**. Готовый проект вы можете найти ниже, в прикрепленном к статье примере.*

И последним визуальным элементом будет размер файла, если конечно текущий объект является таковым.

```
// размер файла, если это файл
if (type == 1)
{
  this.Children.Add(new TextBlock() { Text = String.Format("{0:##,###,##0.00} Кб", size),
HorizontalAlignment = System.Windows.HorizontalAlignment.Right, Margin = new Thickness(8,
0, 0, 0), FontSize = 9, Foreground = new SolidColorBrush(Color.FromArgb(255, 128, 128,
128)) });
}
```

С конструктором мы разобрались. Теперь перейдем к управляющим методам (открыть, переименовать, удалить), которые вызываются при клике левой кнопкой мышки на соответствующей иконке-кнопке.

Метод **Open** должен открывать файлы в новом окне браузера пользователя, либо осуществлять переход по папкам, т.е. отправлять запрос удаленному серверу на получение списка файлов из каталога, для которого вызывается этот метод. Но элемент списка не имеет и не будет иметь методов для взаимодействия с удаленным сервером, весь функционал манипуляции с файлами будет реализован в самом списке, т.е. родителе (**Parent**) текущего элемента. Так что метод **Open** элемента для папок просто вызывает соответствующий метод из списка, в котором он (элемент списка) находится. Речь о методах списка пойдет чуть ниже, когда мы доберемся до его (списка) создания.

```
public void Open()
{
  if (this.ItemType == 1)
  {
    // файл, открываем в новом окне
    HtmlPage.PopupWindow(new Uri(this.FileUrl), "_blank", null);
  }
  else if (this.ItemType == 0)
  {
    // папка,
    // добавляем имя папки к текущему пути
    if (!((FileList)this.Parent).Path.EndsWith("/")) ((FileList)this.Parent).Path += "/";
    ((FileList)this.Parent).Path += this.FileName;
    // обновляем список
    ((FileList)this.Parent).UpdateFileList();
  }
  else if (this.ItemType == -1)
  {
    // папка верхнего уровня,
    // удаляем последнюю папку из текущего пути
    string[] arr = ((FileList)this.Parent).Path .Split("/".ToCharArray());
    Array.Resize(ref arr, arr.Length - 1);
    ((FileList)this.Parent).Path  = String.Join("/", arr);
    // обновляем список
    ((FileList)this.Parent).UpdateFileList();
  }
}
```

Аналогично дела обстоят и с методом **Delete**, который запрашивает у пользователя разрешение на удаления файла/папки, и в случае утвердительного ответа вызывает соответствующий метод класса списка файлов.

```
public void Delete()
{
  if (this.ItemType == 0)
  {
    if (MessageBox.Show(String.Format("Вы действительно хотите удалить папку \"{0}\" и
все вложенные файлы?\r\n\r\nВосстановить данные после удаления будет невомзожно.",
this.FileName), "Удалить?", MessageBoxButton.OKCancel) == MessageBoxResult.OK)
    {
      ((FileList)this.Parent).DeleteItem(this);
    }
  }
  else if (this.ItemType == 1)
  {
    if (MessageBox.Show(String.Format("Вы действительно хотите удалить файл \"{0}\"?\r\n\
r\nВосстановить данные после удаления будет невомзожно.", this.FileName), "Удалить?",
MessageBoxButton.OKCancel) == MessageBoxResult.OK)
    {
      ((FileList)this.Parent).DeleteItem(this);
    }
```

```
  }
}
```

Функционал изменения имени элемента (файла/папки) будет более сложным, хотя принципиально таким же, как и выше описанные методы. Чтобы пользователь имел возможность изменять имя файла/папки, ему нужно предоставить поле для ввода нового имени (**TextBox**). Поэтому процесс редактирования будет начинаться с метода **EditStart**, который выполнит замену нередактируемого элемента **TextBlock** на **TextBox**. Чтобы в последующем избежать путницы, в класс **FileItem** необходимо добавить свойство **IsEdit** типа **bool**, которое будет указывать, находится элемент списка в режиме редактирования или нет.

```
public void EditStart()
{
  // если элемент уже находится в режиме редактирования, то выходим
  if (this.IsEdit) return;
  // удаляем название файла/папки
  this.Children.RemoveAt(1);
  // добавляем TextBox для ввода нового названия папки/файла
  this.Children.Insert(1, new TextBox() { Text = this.FileName, Margin = new Thickness(2,
0, 0, 0) });
  // при потере текстовым элементом фокуса завершаем редактирование
  ((TextBox)this.Children[1]).LostFocus += new RoutedEventHandler(EditTextBox_LostFocus);
  // выбираем весь текст, если это папка
  if (this.ItemType == 0)
  {
    ((TextBox)this.Children[1]).SelectAll();
  }
  else
  {
    // выбираем только имя файла, без расширения (прям как в Windows 7)
    ((TextBox)this.Children[1]).SelectionStart = 0;
    ((TextBox)this.Children[1]).SelectionLength = this.FileName.LastIndexOf(".");
  }
  // фокусируемся на текстовом поле (TextBox)
  ((TextBox)this.Children[1]).Focus();
  // ставим отметку, что элемент находится в режиме редактирования
  this.IsEdit = true;
}
```

Процесс редактирования может быть завершен двумя методами. Если пользователь откажется от редактирования, например, нажмет на клавишу **Esc**, то элемент должен быть возвращен в исходное состояние, без изменения имени. Делать это будет метод **EditCancel**, который удалит **TextBox** с именем файла/папки и поставит на его место **TextBlock**.

```
public void EditCancel()
{
  // удаляем текстовое поле с названием файла/папки
  this.Children.RemoveAt(1);
  // добавляем не редактируемую надпись со старым названием папки/файла
  this.Children.Insert(1, new TextBlock() { Text = this.FileName, Margin = new
Thickness(2, 0, 0, 0) });
  // ставим отметку, что элемент в данный момент не редактируется
  this.IsEdit = false;
  // возвращаем фокус списку
  ((FileList)this.Parent).Focus();
}
```

А второй метод будет производить изменение имени файла/папки и, в случае успеха, приводить элемент в исходное состояние. Запрос на изменение имени файла/папки отправляется из класса списка.

```
public void EditComplete()
{
  // удаляем обработчик потери фокуса у текстового поля, чтобы не делать отправку запроса на изменение имени
  ((TextBox)this.Children[1]).LostFocus -= EditTextBox_LostFocus;
  // получаем новое имя
  this.NewName = ((TextBox)this.Children[1]).Text;
  // отправляем запрос на изменение имени
  ((FileList)this.Parent).SetNewName(this);
}
```

Если вы заметили, в процедуре **EditStart**, текстовому полю добавляется обработчик потери фокуса, в котором вызывается метод **EditComplete**. Нечто подобные вы можете наблюдать в проводнике **Windows**.

```
private void EditTextBox_LostFocus(object sender, RoutedEventArgs e)
{
  EditComplete();
}
```

Поскольку серверу в запросе на изменение имени файла/папки нужно передавать старое (текущее) имя и новое, то для нового имени необходимо сделать свойство **NewName**.

```
public string NewName { get; set; }
```

Класс FileList (список файлов)

Элемент списка у нас готов, теперь можно приступить к реализации самого списка, который будет называться **FileList**.

```
public class FileList : ListBox
{
}
```

Поскольку списку потребуется отправлять запросы на удаленный сервер, он должен знать, куда именно их отправлять, т.е. содержать адрес странички сервера (**Url**). Помимо этого, из обязательных параметров, которые ожидает сервер, нужно передать каталог, который в текущий момент просматривает пользователь (**Path**).

```
private string _Path = "/";
public string Path
{
  get { return _Path; }
  set
  {
    _Path = value;
    if (String.IsNullOrEmpty(_Path)) _Path = "/";
  }
}

public string Url { get; set; }
```

В процесс выполнения запросов к удаленному серверу могут возникать ошибки, которые нужно будет выводить пользователю. Проще всего для этого использовать стандартный **MessageBox**. Однако, поскольку в большинстве случаев код у нас выполняться в отдельных от основного потоках, обычный вызов **MessageBox** может привести к падению приложения. Избежать это можно путем передачи управления в основной поток методом **BeginInvoke**. Чтобы постоянно не писать громоздкий код, для отображения сообщений об ошибках, сделаем вспомогательную процедуру - **ShowError**.

```
private void ShowError(string msg)
{
  this.Dispatcher.BeginInvoke(() =>
  {
    MessageBox.Show(msg, "Ошибка", MessageBoxButton.OK);
  });
}
```

Основным методом списка будет получение файлов и папок с сервера (**UpdateFileList**). Если бы у нас не было класса **WebHelper**, нам бы пришлось писать три отдельных метода для отправки одного запроса. Со вспомогательным классом мы можем ограничиться всего двумя методами на один запрос и значительно меньшим количеством строк кода.

Примечание. *Метод может быть и один, причем и в классе **WebHelper**, где для выполнения одного запроса используются три отдельных метода. В данной статье я стараюсь писать расширенный код, чтобы вам было проще в нем разобраться.*

```
public void UpdateFileList()
{
  WebHelper w = new WebHelper(this.Url);
  w.Queries.Add("cmd", "get");
  w.Queries.Add("path", _Path);
  w.Execute(UpdateFileListResult);
}
```

Результат выполнения запроса на получение списка файлов будет передан в процедуру **UpdateFileListResult** .

```
private void UpdateFileListResult(string stat, string msg, bool allowUp, JsonValue data, 
object tag)
{
  if (stat == "ok")
  { // очищаем список
    this.Dispatcher.BeginInvoke(() =>
    {
      this.Items.Clear();
    });
    // если сервер говорит, что можно подняться на уровень выше, то первым элементом 
добавляем папку верхнего уровня
    if (allowUp)
    {
      AddItem(-1, "...", "", 0);
    }
    // если сервер вернул список папок/файлов, добавляем их
    if (data != null && data.Count > 0)
    {
      foreach (JsonValue itm in data)
      {
```

```
        // добавляем файл в список
        AddItem(itm["type"], itm["name"], itm["url"], itm["size"]);
      }
    }
  }
  else
  {
    ShowError("Ошибка. " + msg);
  }
}
```

Добавление файлов/папок в список производится отдельным методом - **AddItem**. Каждый элемент списка является экземпляром класса **FileItem**, который мы с вами ранее сделали.

```
private void AddItem(int type, string name, string url, double size)
{
  this.Dispatcher.BeginInvoke(() =>
  {
    FileItem itm = new FileItem(type, name, url, size);
    this.Items.Add(itm);
  });
}
```

Аналогично сделаем метод для отправки запроса на удаление файла/папки (**DeleteItem**). Этот метод принимает экземпляр элемента списка (**FileItem**), который нужно удалить. Ссылку на удаляемый элемент (**FileItem**) мы передадим в свойство **Tag** экземпляра класса **WebHelper**.

```
public void DeleteItem(FileItem itm)
{
  WebHelper w = new WebHelper(this.Url);
  w.Queries.Add("cmd", "delete"); // команда delete
  w.Queries.Add("path", _Path);
  w.Queries.Add("name", itm.FileName);
  w.Tag = itm;
  w.Execute(DeleteItemResult);
}
```

В случае успешного удаления файла с сервера, мы можем удалить элемент из списка напрямую, взяв его из свойства **tag**. Если бы мы не передали ссылку на удаляемый элемент, нам бы пришлось делать отдельный запрос на обновление списка файлов/папок (**UpdateFileList**).

```
private void DeleteItemResult(string stat, string msg, bool allowUp, JsonValue data, object tag)
{
  if (stat == "ok")
  {
    // удаляем элемент из списка
    this.Dispatcher.BeginInvoke(() =>
    {
      FileItem itm = tag as FileItem;
      this.Items.Remove(itm);
    });
  }
  else
  {
    ShowError("Ошибка. " + msg);
  }
}
```

Метод для отправки запроса на изменение имени файла/папки (**SetNewName**) похож на **DeleteItem**, затем лишь исключением, что после успешного выполнения команды, элемент списка не удаляется, а просто меняется имя и происходит его вывод из режима редактирования (**EditCancel**).

```
public void SetNewName(FileItem itm)
{
  WebHelper w = new WebHelper(this.Url);
  w.Queries.Add("cmd", "rename"); // команда rename
  w.Queries.Add("path", _Path);
  w.Queries.Add("oldName", itm.FileName);
  w.Queries.Add("newName", itm.NewName);
  w.Tag = itm;
  w.Execute(SetNewNameResult);
}
private void SetNewNameResult(string stat, string msg, bool allowUp, JsonValue data, 
object tag)
{
  if (stat == "ok")
  {
    // меняем имя файла у текущего элемента
    this.Dispatcher.BeginInvoke(() =>
    {
      FileItem itm = tag as FileItem;
      itm.FileName = itm.NewName; // меняем старое имя на новое
      itm.FileUrl = itm.FileUrl.Substring(0, itm.FileUrl.LastIndexOf("/") + 1) + 
itm.FileName; // меняем url
      // и завершаем редактирование
      itm.EditCancel();
    });

  }
  else
  {
    ShowError("Ошибка. " + msg);
  }
}
```

Еще у нас будет метод для создания новой папки, о котором я чуть не забыл, ибо в самом списке он не используется, но логичней реализовать его рядом с остальными. Для вызова метода **CreateDirectory** мы будем использовать внешнюю кнопку на странице, на которой будет размещен список.

```
public void CreateDirectory(string name)
{
  WebHelper w = new WebHelper(this.Url);
  w.Queries.Add("cmd", "newdir"); // команда newdir
  w.Queries.Add("path", _Path);
  w.Queries.Add("name", name);
  w.Execute(CreateDirectoryResult);
}
private void CreateDirectoryResult(string stat, string msg, bool allowUp, JsonValue data, 
object tag)
{
  if (stat == "ok")
  {
    // ответ сервера положительный, значит папка создана
    // обновляем список
    UpdateFileList();
  }
  else
```

```
  {
    ShowError("Ошибка. " + msg);
  }
}
```

Вот мы и подобрались к самому главному – загрузке файлов на сервер. В общем-то, процесс взаимодействия с удаленным сервером принципиально ничем не отличается от выше описанных методов. Но перед отправкой файла нужно будет сделать запрос на проверку возможности загрузить файл на сервер (**check**). Если сервер ответит положительно (**ok**), файл можно будет отправлять. Необходимые методы можно прописать в классе **FileList**, однако если потребуется отправлять на сервер одновременно множество файлов, это может быть не совсем удобно. Поэтому сделаем для отправки файлов отдельный класс, назовем его **UploadItem**.

```
public class UploadItem
{
}
```

Класс должен содержать событие **Complete**, которое будет происходить после завершения взаимодействия с сервером. Благодаря этому событию, список файлов сможет определить, какие файлы были загружены и сколько еще осталось.

```
public event EventHandler Complete;
```

Событие **Complete** будет происходить и при успешном выполнении операции и при возникновении ошибок. Чтобы имеет возможность определить, в каком состоянии находится загружаемый на сервер файл, в классе для этого нужно реализовать пару свойств **State** и **Message**. Свойство **State** будет содержать одно из доступных значений перечисления **StateList**: **OK** – файл успешно загружен, **Error** – ошибка, **Wait** – файл находится в очереди для отправки на сервер. Свойство **Message** будет содержать сообщение об ошибке, если **State** равно **Error**.

```
public enum StateList
{
  OK,
  Error,
  Wait
}

public StateList State { get; set; }
public string Message { get; set; }
```

Также класс должен содержать тело и имя загружаемого файла. Для этого потребуется еще два свойства.

```
public string FileName { get; set; }
public Stream FileStream { get; set; }
```

Для выполнения запросов классу необходимо знать адрес страницы и путь к каталогу, в который необходимо сохранить файл на сервере.

```
private string _Path = String.Empty;
private string _Url = String.Empty;
```

Информация о файле, **url** и путь к каталогу будут передаваться в класс при его инициализации.

```
public UploadItem(FileInfo f, string url, string path)
{
  _Path = path;
  _Url = url;
  this.FileName = f.Name;
  this.FileStream = f.OpenRead();
}
```

Также в классе нужно реализовать метод, который будет запускать процесс отправки файла на сервер, назовем его **Run**. Этот метод будет отправлять запрос на проверку возможности загрузить файл на сервер (**check**).

```
public void Run()
{
  try
  {
    WebHelper w = new WebHelper(_Url);
    w.Queries.Add("cmd", "check"); // команда check
    w.Queries.Add("path", _Path);
    w.Queries.Add("name", _FileName);
    w.Execute(CheckNameResult);
  }
  catch (Exception ex)
  {
    _State = StateList.Error;
    _Message = ex.Message;
    Complete(this, null);
  }
}
```

При положительном ответе (**ok**), файл будет отправлен на сервер.

```
private void CheckNameResult(string stat, string msg, bool allowUp, JsonValue data,
object tag)
{
  try
  {
    if (stat == "ok")
    {
      WebHelper w = new WebHelper(_Url);
      w.Queries.Add("cmd", "upload"); // команда upload
      w.Queries.Add("path", _Path);
      w.Queries.Add("file1", _FileName, _FileStream);
      w.Execute(UploadResult);
    }
    else
    {
      _State = StateList.Error;
      _Message = msg;
      Complete(this, null);
    }
  }
  catch (Exception ex)
  {
    _State = StateList.Error;
    _Message = ex.Message;
    Complete(this, null);
  }
}
```

Результат загрузки файла передается в свойства **State** и **Message**, после чего инициализируется событие **Complete**.

```
private void UploadResult(string stat, string msg, bool allowUp, JsonValue data, object tag)
{
  if (stat == "ok")
  {
    // файл успешно загружен
    _State = StateList.OK;
  }
  else
  {
    _State = StateList.Error;
    _Message = msg;
  }
  Complete(this, null);
}
```

Класс **UploadItem** является элементом очереди на отправку файла на сервер, т.е. он будет частью коллекции. Очередь будет формироваться в списке (**FileList**). Для этого создадим на уровне класса **FileList** локальную переменную.

```
private List<UploadItem> _UploadFiles = null;
```

Файлы, которые нужно загрузить мы будем принимать напрямую из проводника пользователя, при их перетаскивании в список. Для этого необходимо разрешить элементу обрабатывать сообщения системы о перетаскивании (**AllowDrop**) и реализовать обработчик события **Drop**. Сделать это можно в конструкторе.

```
public FileList()
{
  this.AllowDrop = true;
  this.Drop += new DragEventHandler(FileList_Drop);
}
```

В обработчике события перетаскивания нужно получить массив файлов и поместить их в очередь на загрузки **_UploadFiles**. После чего произвести загрузки методом **Upload**.

```
private void FileList_Drop(object sender, DragEventArgs e)
{
  FileInfo[] files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];
  foreach (FileInfo f in files)
  {
    this.AddUploadItem(f);
  }
  this.Upload();
}
```

Для добавления файла в очередь у нас будет использоваться вспомогательный метод **AddUploadItem**.

```
public void AddUploadItem(FileInfo f)
{
  if (_UploadFiles == null) _UploadFiles = new List<UploadItem>();
  UploadItem itm = new UploadItem(_UploadFiles.Count, f, this.Url, this.Path);
  itm.Complete += new EventHandler(UploadItem_Complete);
  _UploadFiles.Add(itm);
}
```

Как видите в приведенном выше коде, при добавлении файлов в очередь, к каждому элементу подключается обработчик события **Complete**. В этом событие загруженный файл удаляется из очереди. Когда очередь закончится, будет отправлена команда на обновление списка файлов.

```
private void UploadItem_Complete(object sender, EventArgs e)
{
  this.Dispatcher.BeginInvoke(() =>
  {
    UploadItem itm = sender as UploadItem;
    // удаляем из очереди загруженный файл
    _UploadFiles.Remove(itm);
    if (_UploadFiles.Count == 0)
    {
      UpdateFileList();
    }
  });
}
```

Собственно, в самом методе загрузки (**Upload**) производится цикличный вызов процедуры **Run** для каждого элемента.

```
public void Upload()
{
  if (_UploadFiles == null || _UploadFiles.Count <= 0) return;
  foreach (UploadItem itm in _UploadFiles)
  {
    itm.Run();
  }
}
```

Использование и улучшение списка файлов

Список готов. Компилируйте проект и вы найдете созданный нами список на панели элементов **Visual Studio**.

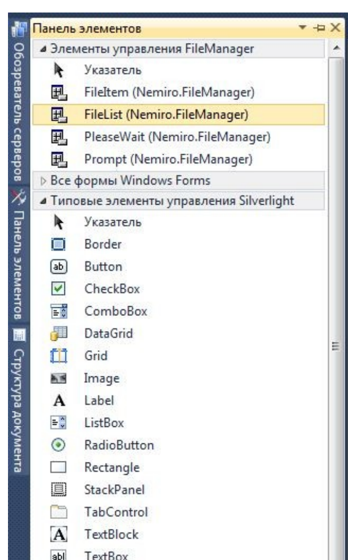

Рис. 1. Расположение созданных элементов на панели элементов **Visual Studio**.

Либо пропишите его вручную на странице **Silverlight**.

```
<my:FileList Height="256" HorizontalAlignment="Left" Margin="12,41,0,0"
x:Name="fileList1" VerticalAlignment="Top" Width="573" Grid.ColumnSpan="2"
Grid.RowSpan="2" Url="http://localhost:58646/FileManagerGateway.ashx" />
```

Обратите внимание, что для корректной работы списка файлов необходимо указать адрес страницы сервера, которая готова обрабатывать команды клиента. В моем случае это http://localhost:58646/FileManagerGateway.ashx, у вас наверняка адрес будет другим. Лучше **Url** указывать программно (а не в **XAML**, как показано выше), например, в конструкторе страницы, на которой размещен список, т.к. использование асинхронных запросов может привести к падению **Visual Studio**.

```
public MainPage()
{
  InitializeComponent();
  fileList1.Url = "http://localhost:58646/FileManagerGateway.ashx";
}
```

Чтобы приложение **Silverlight** имело возможность отправлять запросы серверу, на сервере необходимо разместить файл **clientaccesspolicy.xml**, содержащий соответствующие разрешения.

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Примечание. *Рекомендуется ставить ограничения по домену (**domain**). В приведенном выше примере разрешена обработка удаленных запросов с любых доменов (*).*

Чтобы загрузить файлы и папки с сервера, необходимо вызвать метод **`UpdateFileList`**.

```
fileList1.UpdateFileList();
```

Результат проделанной работы показан на рис. 2.

Рис. 2. Результат проделанной работы.

Для полноты картины, чтобы при двойном клике мышкой осуществлялся переход в указанный каталог или открывался файл, необходимо реализовать соответствующий обработчик в элементе списка (**FileItem**). К сожалению, в **Silverlight** пока такого события нет, так придется изобретать свой велосипед.

```
private DateTime _lastClick = DateTime.Now;
private bool _firstClickDone = false;
private void FileItem_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)
{
  DateTime clickTime = DateTime.Now;
  TimeSpan span = clickTime - _lastClick;
  if (span.TotalMilliseconds > 350 || !_firstClickDone)
  {
    _firstClickDone = true;
    _lastClick = clickTime;
  }
  else
  {
    _firstClickDone = false;
    Open();
  }
}
```

Добавить обработчик можно в конструкторе класса.

```
public FileItem(int type, string name, string url, double size)
{
  this.MouseLeftButtonUp += new MouseButtonEventHandler(FileItem_MouseLeftButtonUp);
  // в дополнение к тому коду, который уже есть
}
```

Также можно сделать обработчик нажатия кнопок клавиатуры, но уже на уровне списка (**FileList**).

```
private void FileList_KeyUp(object sender, KeyEventArgs e)
{
  if (((FileList)sender).SelectedItem == null) return;
  FileItem itm = ((FileList)sender).SelectedItem as FileItem;
  // если нажата клавиша Enter и элемент не находится в режиме редактирования
  if (e.Key == Key.Enter && !itm.IsEdit)
  { // открываем файл/папку
    itm.Open();
  }
  // если нажата клавиша Enter и элемент находится в режиме редактирования
  else if (e.Key == Key.Enter && itm.IsEdit)
  { // открываем файл/папку
    itm.EditComplete();
  }
  // если нажата клавиша F2, элемент можно редактировать и он не находится в режиме
редактирования
  else if (e.Key == Key.F2 && itm.CanEdit && !itm.IsEdit)
  { // запускаем режим редактирования
    itm.EditStart();
  }
  // если нажата клавиша Esc и элемент находится в режиме редактирования
  else if (e.Key == Key.Escape && itm.IsEdit)
  {
    // отменяем редактирование
    itm.EditCancel();
  }
  // если нажата клавиша Delete и элемент не находится в режиме редактирования
  else if (e.Key == Key.Delete && !itm.IsEdit)
  {
```

```
    // отправляем запрос на удаление элемента
    itm.Delete();
  }
  // если нажата клавиша F5
  else if (e.Key == Key.F5)
  {
    // обновляем список файлов
    UpdateFileList();
  }
}
```

Аналогично, обработчик события можно подключить в конструкторе класса.

```
public FileList()
{
  this.KeyUp += new KeyEventHandler(FileList_KeyUp);
  // в дополнение к тому коду, который уже есть
}
```

Чтобы список файлов загружался автоматически, сразу после загрузки приложения, необходимо добавить обработчик события **Loaded.**

```
this.Loaded += new RoutedEventHandler(FileList_Loaded); // в конструктор FileList
// и сам обработчик
private void FileList_Loaded(object sender, RoutedEventArgs e)
{
  UpdateFileList();
}
```

При желании, на странице со списком файлов можно разместить кнопку, которая позволит открыть диалоговое окно для выбора файлов, которые требуется загрузить на сервер. Принцип работы с файлами будет таким же, как и при перетаскивании.

```
private void btnUploadFile_Click(object sender, RoutedEventArgs e)
{
  // показываем диалог выбора файлов
  OpenFileDialog op = new OpenFileDialog() { Filter = "Изображения|
*.jpg;*.jpeg;*.gif;*.png|Документы|*.txt;*.rtf;*.doc;*.docx;*.xls;*.xlsx;*.pdf|
Аудиофайлы|*.wav;*.mp3;*.wma|Видеофайлы|*.avi;*.mpg;*.mov;*.wmv;*.mp4;*.mpeg|Все файлы|
*.*", Multiselect = true };
  if (op.ShowDialog() != true) return; // пользователь отказался выбирать файлы, выходим

  // готовим выбранные файлы к отправке на сервер
  foreach (FileInfo f in op.Files)
  {
    fileList1.AddUploadItem(f);
  }

  // отправляем файлы
  fileList1.Upload();
}
```

Если вы помните, у нас есть метод (о котором я чуть не забыл), который позволяет создавать новые папки на сервере. Для его использования, на странице можно разместить еще одну кнопку, **btnCreateDir**.

```
private void btnCreateDir_Click(object sender, RoutedEventArgs e)
{
  Prompt myPrompt = new Prompt("Ввод", "Введите имя папки:");
  myPrompt.Closed += (s, args) =>
  {
    if (!myPrompt.DialogResult.Value) return;
    fileList1.CreateDirectory(myPrompt.InputValue);
  };
  myPrompt.Show();
}
```

Для ввода имени папки я сделал отдельное дочернее окошко - **Prompt**.

```
<controls:ChildWindow x:Class="Nemiro.FileManager.Prompt"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:controls="clr-
namespace:System.Windows.Controls;assembly=System.Windows.Controls"
           Width="400" Height="140"
           Title="Prompt"
xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk">
    <Grid x:Name="LayoutRoot" Margin="2">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Button x:Name="CancelButton" Content="Отмена" Click="CancelButton_Click"
Width="75" Height="23" HorizontalAlignment="Right" Margin="0,12,0,0" Grid.Row="1" />
        <Button x:Name="OKButton" Content="OK" Click="OKButton_Click" Width="75"
Height="23" HorizontalAlignment="Right" Margin="0,12,79,0" Grid.Row="1" />
    <sdk:Label Height="19" HorizontalAlignment="Left" Margin="0,12,0,0" Name="label1"
VerticalAlignment="Top" Width="378" />
    <TextBox Height="23" HorizontalAlignment="Left" Margin="0,37,0,0" Name="textBox1"
VerticalAlignment="Top" Width="378" />
  </Grid>
</controls:ChildWindow>
```

```
  public partial class Prompt : ChildWindow
  {

    /// <summary>
    /// Возвращает либо устанавливает значение для поля ввода
    /// </summary>
    public string InputValue
    {
      get
      {
        return textBox1.Text;
      }
      set
      {
        textBox1.Text = value;
      }
    }

    public Prompt(string title, string promptText)
    {
      InitializeComponent();

      this.Title = title;
      label1.Content = promptText;
    }
```

```csharp
    private void OKButton_Click(object sender, RoutedEventArgs e)
    {
      if (String.IsNullOrEmpty(textBox1.Text))
      {
        MessageBox.Show("Необходимо ввести значение.", "Ошибка", MessageBoxButton.OK);
        return;
      }
      this.DialogResult = true;
    }

    private void CancelButton_Click(object sender, RoutedEventArgs e)
    {
      this.DialogResult = false;
    }
  }
```

Послесловие

Сделанный нами файловый менеджер можно использовать в самостоятельных проектах **Silverlight**, либо размещать в виде дополнительного компонента на страницах **ASP.NET (WebForm/ MVC)**. Аналогично и с классом **WebHelper**, который запросто можно вынести за рамки этого проекта и использовать для других целей.

Очевидно, что многие описанные в статье классы можно объединить. Я этого делать не стал, чтобы вам было проще во всем разобраться, да и мне не стереть остатки клавиш, объясняя, что к чему.

В этой статье я не стал рассматривать варианты разграничения доступа к файлам, статья и без того получилась довольно объемной. Но в идеале, конечно, необходимо ограничивать доступ пользователей к файлам, создать механизмы авторизации и более тщательно проверять загружаемые файлы, вести учет. Если говорить о реализации, то можно проводить авторизацию пользователя прямо в приложении **Silverlight**. Либо, если файловый менеджер размещен на странице в проекте **ASP.NET**, где пользователь уже авторизирован, можно передавать в приложение данные пользователя через параметры, которые будут передаваться обратно в запросах к шлюзу, а еще лучше использовать секретную подпись, что повысит безопасность всей системы.

В примере, который вы можете скачать ниже, помимо всего прочего, реализован вывод прогресса, благодаря добавлению двух событий в список файлов, определяющих начало процесса и его завершение. Также в примере лучше отработан функционал оповещения об ошибках. Класс **Gateway** я вывел в отдельную, независимую, библиотеку (dll), которую можно подключать к проектам **ASP.NET**. Код проекта детально прокомментирован. При работе с проектом, рекомендуется использовать **Visual Studio 2010** с редакцией отличной от **Express**, т.к. решение состоит из 4 связанных проектов. Если у вас **Express** издание, то для проверки работы примера, вам потребуется отдельно открыть и запустить (**F5**) один из web-проектов (**WebForms** или **MVC**) и проект **Silverlight**, указав элементу **FileList** адрес на страницу шлюза запущенного web-проекта. Пример использования приложения **Silverlight** находится в проекте **FileManager.Web** (**WebForms**) на странице **FileManagerTestPage.aspx**.

Несмотря на то, что файловый менеджер, в принципе, готов, ему есть куда расти и чем наращивать функционал.

Если у вас возникнут какие-либо вопросы, обращайтесь на форум Kbyte.Ru.

--
Алексей Немиро
05 января 2012 года