Задание №9. Сериализация данных

Сериализация

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Эта операция предназначена для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется ***десериализацией***.

Применение сериализации:

- Отправка объекта в удаленное приложение с помощью веб-службы.
- Передача объекта из одного домена в другой.
- Передача объекта через брандмауэр в виде строки JSON или XML.
- Хранение сведений о безопасности и пользователях между приложениями.

Есть несколько способов представить информацию:

- бинарное представление.
- представление в текстовом формате xml.
- представление в текстовом формате json.

Бинарное представление является самым эффективным с точки зрения хранения, сериализации и десериализации структур данных – фактически, при бинарной сериализации происходит физическое копирование фрагмента оперативной памяти, например, в файл. Поскольку в файл записывается фрагмент памяти, то последующее восстановление объекта (десериализация) заключается в копировании этого фрагмента памяти обратно в оперативную память компьютера с минимальными преобразованиями типов данных.

Сериализация в текстовые форматы xml или json удобнее с точки зрения того, что разработчик может просмотреть данные в текстовом редакторе и самостоятельно убедиться в их правильности. Пример объекта Person, представленного в формате json:

```
{
  "Surname": "Smirnov",
  "Name": "Yuriy",
  "Phone": "+71112223344"
```

```
}
```

Однако, представление некоторых типов данных в текстовом формате может быть неэффективным, например:

```
"3,14159265358979"
```

Число Пи, являясь вещественным числом, занимает в оперативной памяти 8 байт (а, значит, и 8 байт при бинарной сериализации), в то время как текстовое представление числа Пи с высокой точностью займет 16 символов (32 байта), что, очевидно, неэффективно для хранения данных. Кроме того, текстовое представление требует хранение информации о названии поля, что также будет занимать дополнительный объем памяти. Третья проблема – во время сериализации необходимо написать алгоритм преобразования числа в строку, а при десериализации – преобразования строки в число. Любое преобразование строки в число займет больше времени процессора, чем обычное копирование бинарного представления из файла в оперативную память. Текстовое представление данных проигрывает бинарному в производительности и объёме занимаемой памяти.

Несмотря на это, именно передачу в формате xml или json предпочитают использовать для обмена данными между сервером и клиентом в клиент-серверных или веб-приложениях.

В C# есть готовые решения для каждого из перечисленных способов:

- BinaryFormatter
- XmlSerializer
- JsonSerializer

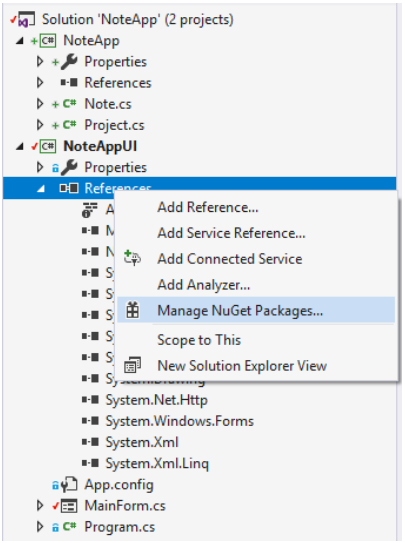
Подробное описание работы данных классов с примерами кода можно найти на сайте docs.microsoft.com.

Далее в разрабатываемом приложении вам потребуется реализовать функцию сохранения и загрузки данных пользователя между запусками приложения с помощью механизма сериализации. Задание предполагает реализацию на основе json-формата. Несмотря на наличие готовых решений работы с xml и json, стандартные реализации XmlSerializer и JsonSerializer ограничены в своей функциональности и требуют написания большого количества дополнительного кода. Для сохранения данных в формате json существует сторонняя библиотека Newtonsoft JSON.NET, о подключении и использовании которой пойдет речь в следующих разделах.

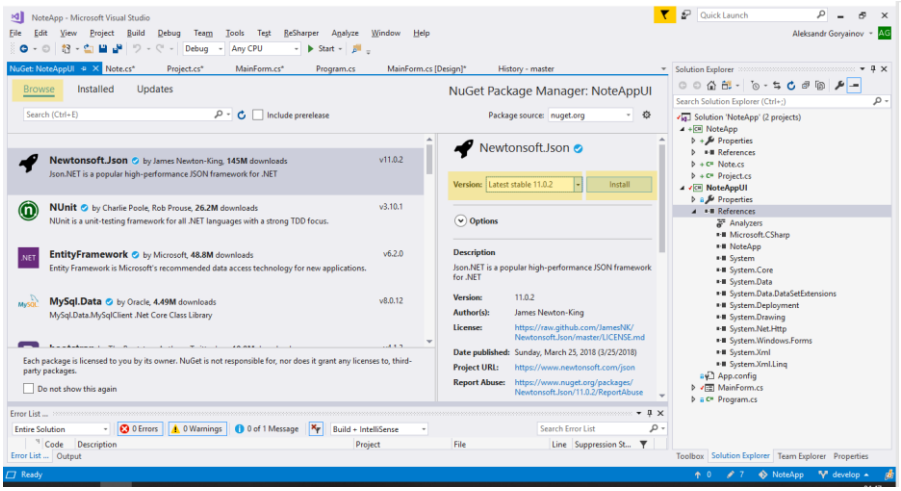
Подключение сторонних библиотек с помощью менеджера пакетов NuGet

Стандартный способ подключения сторонних библиотек обладает своими недостаткам. Например, если у сторонней библиотеки выходит обновление, то переподключение новой версии библиотеки приходится делать вручную. Это особенно важно в случае критических обновлений, обновлений библиотек связанных с безопасностью данных. Также иногда возникает потребность в откате версий используемых библиотек на более ранние. Для того, чтобы упростить перечисленные задачи, был разработан новый механизм подключение .NET-сборок – менеджер подключения пакетов NuGet.

Для подключения сборки в проект с помощью NuGet, кликните правой кнопкой по пункту References целевого проекта в окне Solution Explorer, и выберите в появившемся контекстом меню пункт Manage NuGet Packages («управление пакетами NuGet»):

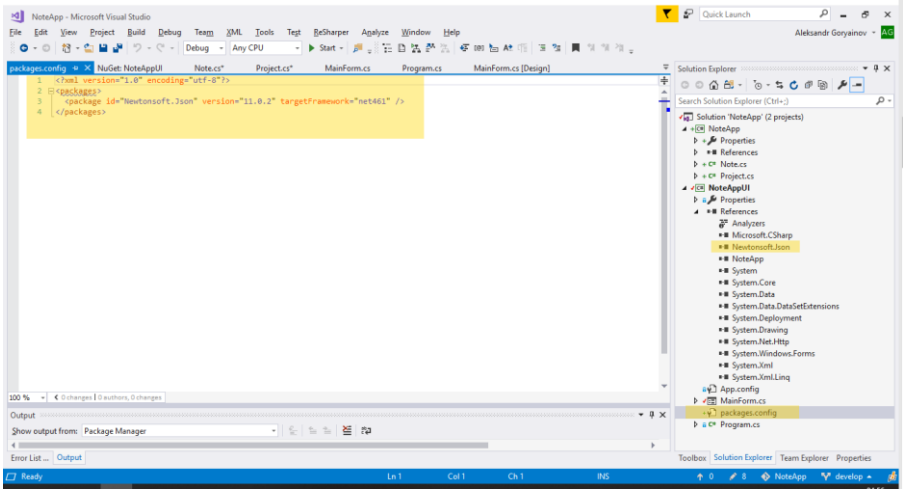


После выбора пункта меню появится вкладка менеджера пакетов (см. рисунок далее). В данной вкладке можно установить новые пакеты (вкладка Browse), так и управлять уже подключенными сборками (вкладка Installed). Для подключения сторонних сборок необходимо подключение к Интернету. Для каждой сборки перед установкой вы можете выбрать нужную версию (в том числе, ранние стабильные версии), а также прочитать описание. Скачивание библиотеки и установка может занять некоторое время.



Вкладка менеджера пакетов NuGet

После успешной установки библиотеки, ссылка добавится в проект, а в корне проекта появится файл packages.config:



packages.config хранит ссылки всех подключенных через NuGet пакетов

В данном файле хранится информация о подключенных через NuGet сборках, поэтому удалять данный файл не нужно.

Небольшое примечание к работе NuGet: при открытии вашего решения на другом компьютере, Visual Studio изначально не сможет найти подключенные сборки и обозначит библиотеки NuGet как отсутствующие (появятся соответствующие ошибки компилятора, а сами сборки будут отмечены желтыми значками). Однако во время первой компиляции решения, среда разработки автоматически установит требуемые библиотеки, и программа будет работать корректно.

Менеджер NuGet автоматически вас предупредит в случае появления обновлений к используемым библиотекам, обновление библиотек можно будет выполнить в нажатие одной кнопки.

Библиотека сериализации Newtonsoft JSON.NET

Библиотека JSON.NET от компании Newtonsoft предназначена для сериализации и десериализации объектов в текстовый формат json. Несмотря на наличие в .NET нескольких стандартных механизмов сериализации, библиотека от компании Newtonsoft получила более широкое распространение. Преимуществом сторонней библиотеки является возможность сериализации стандартных коллекций (массивов, списков и словарей), также сериализация объектов, хранящихся интерфейсных переменных.

Сериализация в формате json используется для передачи данных между частями системы (например, клиент-серверных приложений или распределенных систем), сохранения данных о самой системе, представления конфигурационных файлов.

Библиотека Newtonsoft JSON.NET доступна для подключения в проект через средства NuGet. В случае, если подключить библиотеку через NuGet не представляется возможным, библиотека может быть скачана с официального сайта и подключена как обычная dll-сборка.

Предположим, что у нас есть класс `Product`, описывающий товар интернет-магазина:

```
public class Product
{
    public string Name { get; set; }

    public double Price { get; set; }

    public string[] Sizes { get; set; }
}
```

И нам необходимо сериализовать экземпляр данного класса. Для начала создадим объект, который будем сериализовывать:

```
var product = new Product();
product.Name = "T-Shirt";
product.Price = 1100.0;
product.Sizes = new string[]
{
    "S",
    "M",
```

```
    "L",
    "XL"
};
```

Теперь используем библиотеку JSON.NET для сериализации:

```
//Создаём экземпляр сериализатора
JsonSerializer serializer = new JsonSerializer();

//Открываем поток для записи в файл с указанием пути
using (StreamWriter sw = new StreamWriter(@"c:\json.txt"))
using (JsonWriter writer = new JsonTextWriter(sw))
{
    //Вызываем сериализацию и передаем объект, который хотим
сериализовать
    serializer.Serialize(writer, product);
}
```

Результатом работы кода будет файл json.txt следующего содержания:

```
{
  "Name": "T-Shirt",
  "Price": 1100.0,
  "Sizes": [
    "S",
    "M",
    "L",
    "XL"
  ]
}
```

Аналогично выполняется десериализация:

```
//Создаём переменную, в которую поместим результат
десериализации
Product product = null;

//Создаём экземпляр сериализатора
JsonSerializer serializer = new JsonSerializer();

//Открываем поток для чтения из файла с указанием пути
```

```
using (StreamReader sr = new StreamReader(@"c:\json.txt"))
using (JsonReader reader = new JsonTextReader(sr))
{
    //Вызываем десериализацию и явно преобразуем результат в
целевой тип данных
    product = (Product)serializer.Deserialize(reader);
}
```

Обратите внимание на необходимость явного преобразования типов при десериализации.

Класс `JsonSerializer` содержит ряд полей-настроек, используемых для форматирования результирующего файла, а также указания настроек самой сериализации. Из наиболее значимых настроек можно выделить:

`JsonSerializer.NullValueHandling` – настройка, указывающая как записывать поля, принимающие значение null.

`JsonSerializer.TypeNameHandling` – настройка, указывающая необходимо ли записывать названия типов данных для сериализуемых объектов. Настройку необходимо использовать при сериализации полей интерфейсного типа данных, а также шаблонных коллекций.

`JsonSerializer.Formatting` – настройка, указывающая форматирование в итогом файле. Например, автоматическая табуляция для вложенных типов данных.

JSON.NET может сериализовать объекты любой сложности. Если вы сериализуете класс, агрегирующий другие типы данных, то все агрегируемые объекты также будут сериализованы. Однако при сериализации стоит избегать сериализации крупных объектов (например, сериализовать экземпляр главного окна программы MainForm будет плохой идеей), а также избегать сериализации объектов с цикличными ссылками (когда объект A агрегирует объект B, в котором хранится ссылка на объект A).

Любая сериализация предъявляет особые требования к конструкторам класса. В частности, если в сериализуемом классе (или агрегируемых им классах) присутствует несколько конструкторов, то необходимо с помощью специального атрибута [JsonConstructor] обозначить тот конструктор, который должен вызываться при десериализации объектов.

Если вы хотите выполнить сериализацию или десериализацию без использования потоков (например, без прямой записи в файл), вы можете использовать класс статический класс JsonConvert, хранящий методы по преобразованию объектов в json-строки и обратно без работы с файлами или иными потоками.

Подробная документация по работе с библиотекой находиться на официальном сайте компании. На сайте также есть обширные примеры исходного кода, и форум с обратной связью, где можно задать вопросы по работе библиотеки разработчикам.

Реализация сериализации в приложении

В предыдущем разделе были представлены готовые алгоритмы по сериализации и десериализации. Однако необходимо рассмотреть вопрос того, как внедрить представленные алгоритмы в программу архитектурно.

В приложении необходимо реализовать сериализацию всех пользовательских данных, которые хранятся в объекте типа Project. Можно было поместить методы по сохранению и загрузке файлов в класс Project. Однако сериализация вполне является самостоятельной задачей, которая может быть вынесена в отдельный класс. Реализация в отдельном классе удобна тем, что вся работа со сторонней библиотекой будет сокрыта (инкапсулирована) в отдельном классе, а остальные классы даже не будут знать об их существовании. Это удобно с точки зрения модификации программы – в случае, если потребуется исправить сериализацию или заменить стороннюю библиотеку на другой механизм сохранения данных, изменения коснуться только одного класса, не затронув остальной системы.

Так как класс занимается сериализацией проекта, логично назвать его ProjectSerializer. Класс ProjectSerializer должен реализовывать два метода:

```
public class ProjectSerializer
{
    public void SaveToFile(Project project);
    public Project LoadFromFile();
}
```

Также, нужно передавать информацию о том, в какой файл выполнять сохранение, и из какого файла загружать проект. Имя файла можно передать в качестве дополнительного строкового аргумента в методы сохранения и загрузки, но более удобной реализацией будет хранение имени файла в виде свойства:

```
public class ProjectSerializer
{
    public string Filename {get;set;}
    public void SaveToFile(Project project);
    public Project LoadFromFile();
}
```

Значение свойства можно будет вызывать напрямую внутри методов, что упростит вызов методов в клиентском коде – не надо будет передавать дополнительный аргумент.

Важный момент – где именно должен быть расположен файл с пользовательскими данными.

Сохранение файла на диске C или в папке Program Files невозможно – операционная система не позволит менять файлы в системных папках, чтобы исключить вероятность повреждения ОС. Исключение – программы, запущенные с правами администратора, но большинство пользовательских приложений (включая ваше) не должно иметь прав администратора для своей работы.

Существует две специальные папки для хранения данных пользователя и приложений:

- Мои документы
- AppData

«Мои документы» используются в тех случаях, когда пользователь сам вызывает функцию сохранения файла (например, документ в Word или картинка в Paint), указывает имя файла. Хранить в папке «Мои документы» файлы, которые создаются автоматически без ведома пользователя могут привести к ситуации, что пользователь случайно удалит незнакомый ему файл.

Для хранения данных программы, к которым обычный пользователь не должен иметь доступа, в ОС Windows создана специальная папка AppData. Она размещается по пути:

```
C:\Users\<имя пользователя>\AppData\Roaming
```

Быстро зайти в эту папку можно, если нажать Win + R, и выполнить команду %AppData%.

Папка AppData является скрытой, и рядовой пользователь о существовании этой папки не догадывается. Зайдя в эту папку, вы можете убедиться, что другие установленные приложения создают здесь собственные подпапки для хранения своих данных. Ваша программа также должна хранить свои данные в этой папке.

Возможно, вы обратили внимание, что путь к папке AppData содержит имя пользователя ОС, а значит у разных пользователей путь до этой папки будет различным. Следовательно, путь к папке нельзя задать жестко в коде. Нужно определить путь к папке на компьютере конкретного пользователя. Для этого необходимо вызвать метод стандартного класса Environment:

```
var appDataFolder =
Environment.GetFolderPath(SpecialFolder.ApplicationData);
```

С помощью метода GetFolderPath() вы можете программно получить путь и до других системных папок, таких как Рабочий стол, Мои документы, меню Пуск, Program Files и др., но в контексте нашего приложения нас интересует папка AppData.

К полученному пути необходимо добавить подпапку с вашей фамилией (поп-папка разработчика), и еще одну подпапку с названием вашей программы, и затем имя файла для сохранения. Итоговый путь должен выглядеть примерно так:

```
C:\Users\<имя пользователя>\AppData\Roaming\Goryainov\
NoteApp\userdata.json
```

Создание подпапок разработчика и программы в будущем упрощает удаление пользовательских данных с компьютера при удалении приложения – проще удалить папку программы, чем пытаться удалить файлы из общей папки, где случайно можно удалить данные программ.

Зная, как задать путь к файлу сохранения пользовательских данных, необходимо доработать класс ProjectSerializer:

- По умолчанию (в конструкторе класса) свойство Filename должно инициализироваться путём в папке AppData, как показано выше.
- Методы SaveToFile() и LoadFromFile() должны работать с путём файла, указанным в свойстве Filename.
- Вне зависимости от значения в свойстве Filename, методы SaveToFile() и LoadFromFile() должны проверять, существует ли папка, указанная в свойстве Filename. И, если папка не существует, то методы должны создать папку.
- Методы SaveToFile() и LoadFromFile() должны выполнять обработку исключений, которые могут возникнуть при попытке создания файла, его открытия или десериализации. Распространенные исключения – файл или путь не существует, файл поврежден и не может быть десериализован. В случае возникновения любой из ошибок, метод LoadFromFile() должен возвращать новый пустой объект Project.

Таким образом, если клиент класса ProjectSerializer при сохранении файла захочет поменять путь сохранения, он должен будет перед вызовом присвоить другое значение в свойство Filename, но по умолчанию ProjectSerializer сохраняет и загружает данные в папку AppData. Таким образом, мы выполняет преднастройку класса для его использования, но оставляем гибкость в его использовании, давая возможность сохранения и в другие пути. Возможность сохранения файла по другому пути пригодится при написании юнит-тестов в дальнейшем.

В завершение класс сериализатора можно сделать статическим. Статические классы имеют ряд недостатков, однако в небольшим приложениях, создание сервисных классов допустимо.

Описание задания:

1. Создайте новую ветку в локальном репозитории под названием «features/9_add_project_serializer». Перейдите в новую ветку.
1. Подключите библиотеку Newtonsoft JSON.NET к проекту бизнес-логики, используя менеджер пакетов NuGet.
2. Создайте в проекте бизнес-логики класс ProjectSerializer, если он еще не был создан ранее.
3. В классе ProjectSerializer создайте свойство Filename и методы SaveToFile() и LoadFromFile(), как это описано в разделах выше.
4. Реализуйте инициализацию свойства Filename путем к папке AppData по умолчанию. Не забудьте добавить к пути AppData подпапку с именем разработчика и подпапку с названием программы.
5. В методах сохранения и загрузки проекта реализуйте обработку ошибок на случаи, когда считываемый файл не существует, или когда файл поврежден и выполнить десериализацию невозможно. В случае любых ошибок при десериализации, метод должен возвращать новый пустой объект Project (не null).
6. Убедитесь, что все требования, описанные в разделах выше, выполнены верно.
7. Добавьте xml-комментарии к новому классу.
8. Если вы реализовали класс ProjectSerializer как не статический, добавьте в главное окно MainForm новое поле _projectSerializer и проинициализируйте его новым объектом.
9. Добавьте в код главного окна логику сохранения и загрузки проекта. Загрузка проекта из файла должна вызываться в конструкторе главного окна, после чего главное окно должно инициализироваться на основе данных из загруженного проекта. Загруженный проект должен помещаться в поле _project.
10. Сохранение проекта в файл должно выполняться: а) при закрытии программы; б) при добавлении новых данных пользователем; в) при редактировании данных пользователем; г) при удалении данных пользователем. Так как приложение небольшое, и объём пользовательских данных также предполагается небольшим, в программе можно позволить такие частые сохранения. При разработке высоконагрузочных приложений политика сохранений должна продумываться более тщательно, чтобы не нагружать процессор частыми сохранениями данных.
11. Запустите программу и убедитесь, что пользовательские данные теперь сохраняются при закрытии программы и снова загружаются при её повторном

открытии. Убедитесь, что пользовательские данные действительно сохраняются в папке AppData.

12. Убедитесь, что пользовательские данные сохраняются верно – иногда в файл могут неправильно сохраняться даты (тип DateTime) или неправильно загружаться. Проверить правильность сохранения можно, открыв сохраненный текстовый файл. Правильность загрузки можно проверить, запустив программу.

13. Добавьте xml-комментарии для всех полей и методов в главном окне, которые вы создали при реализации задания. Проверьте правильность именования и отсутствие грамматических ошибок.

14. В Visual Studio перейдите в ветку develop. Затем выполните слияние (merge) ветки features/9_add_project_serializer в ветку develop. Теперь все изменения из ветки с добавленным решением должны переместиться в ветку develop. Убедитесь в этом с помощью GitHub.