[**Программирование сокетов в C#**](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html)

by [Joydip Kanjilal](http://aspalliance.com/author.aspx?uId=58780)   
оригинал статьи: <http://aspalliance.com/1563_Socket_Programming_in_C.all>   
перевод Denis M Kucherov

**Краткое изложение**

В этой статье Joydip продемонстрирует работу сокетов при программировании в C#. Он начнет с краткого ознакомления с сокетами и портами рассмотрит разницу между синхронной и асинхронной передачей данных. Вы узнаете как работать с пространствами имен System.Net и System.Net.Sockets применяя их в простых клиент-серверных приложениях. Статья также снабжена всеми необходимыми примерами кода и результатами их работы в виде скрин-шотов.

**Вступление**

Компьютерная сеть предполагает наличие двух или более компьютеров соединенных вместе посредствам оборудования и программного обеспечения, в целях обмена данными и информацией. Microsoft .NET предоставляет отличную поддержку в написании программ которые могут использовать сокеты для обмена данными между двумя и более компьютерами в сети. По мере чтения данной статьи мы будем узнавать больше о сокетах и портах. В данной статье расматривается то , как мы можем работать с сокетами в .NET, в целях написания программ которые могут работать в сети. Обратите внимание на то что примеры приведенные в статье написаны на C#.

**Понимание Сокетов и Портов**

Сокет, это конечная точка двухстороннего соединения между двумя системами работающих в сети. Когда два или более процесса взаимодействуют через сеть, они взаимодействуют используя сокеты. Порт представляет собой беззнаковое целое число, которое уникально идентифицирует процесс выполняющийся в сети.   
Вот некоторые хорошо известные номера портов охарактеризованные типами сервисов которые их предоставляют:

* HTTP 80
* Telnet 23
* SMTP 25
* SNPP 444
* DNS 53
* FTP (Data) 20
* FTP (Control) 21

Синхронное и асинхронное соединение   
В Microsoft .NET вы имеете поддержку обоих, синхронного и асинхронного соединения при использовании сокетов. Также известный как, блокированный и не блокированный режим работы. Существуют весьма тонкие различия между двумя этими методами. Когда работаем в синхронном режиме, вызов метода блокирует сам себя до тех пор пока операция не будет полностью закончена. В другом режиме работы, т.е. асинхронном режиме, метод возвращается еще до того как время цикла обработки закончилось. Отлично, но что такое время цикла обработки? Под временем цикла обработки подразумевается общее время затраченное потоком до полного его завершения.   
В синхронном режиме взаимодействия, серверное приложение слушает конкретный порт на предмет получения данных от клиента. Поступая так, серверное приложение блокируется (для других клиентов) до тех пор пока не получит данные от клиентского приложения. С другой стороны, во время работы в асинхронном режиме, сервер может обрабатывать множество клиентских запросов одновременно. Обратите внимание, что асинхронные команды использующие сокеты обычно применяются для задач которые требуют больших затрат времени. Типичные примерами таких задач является открытие больших файлов, отправка запросов базе данных с большим объемом данных, подсоединение к удаленному компьютеру, удаленный доступ к ресурсам требующих больших временных затрат.   
Так же учтите, что асинхронные вызовы на самом деле работают в раздельных потоках.   
Обычно программа имеет два вида потоков, программный поток и рабочий поток. Программный поток - это основной поток программы; рабочий поток - это поток работающий в фоне, для обеспечения асинхронных операций.   
Класс Socket в пространстве имен System.Net.Sockets содержит и синхронные и асинхронные методы. К примеру, в то время как Connect() и Recive() методы предназначены для синхронного использования, BeginConnect() и EndConnect() методы, так же как BeginReceive() и EndReceive() методы их асинхронные аналоги.

**Работа с пространствами имен System.Net и System.Net.Sockets**

Класс Socket в пространстве имен System.Net.Sockets используется для работы с сокетами в C#. Заметьте что с каждым сокетом ассоциируется локальные и удаленные конечные точки. Мы рассмотрим этот класс позднее в статье.   
Пространство имен System.Net включает в себя важный клас под названием Dns, который может быть использован для доступа к DNS (Domain Naming Service). Так что же такое DNS? Domain Naiming Service или DNS это сервис имен который используется для именования узлов сети с целью упрощения. Имя домена это текстовое имя которое идентифицирует главный компьютер (host) в сети. DNS сервера хранят DNS адреса совместно с соответствующими им IP адресами.   
Теперь, для того что бы получить хост-имя вашего локального компьютера, используйте следующее:

Console.WriteLine("The host name of the local computer is: " + Dns.GetHostName());

Хост-имя локального компьютера должно отобразиться, как показано ниже.

На верхнем фрагменте кода, GetHostName() статический метод класса Dns. Вот пример программы которая илюстрирует то, как вы можите отобразить IP адрес сайта [www.hotmail.com](http://www.hotmail.com/) используя этот класс.

**Отображение IP адресов сервера Hotmail используя сокеты**

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace TestSocket

{

class Program

{

static void Main(string[]args)

{

try

{

IPHostEntry IPHost = Dns.GetHostEntry("www.hotmail.com");

IPAddress[]ipAddress = IPHost.AddressList;

StringBuilder strIpAddress = new StringBuilder();

for (int i = 0; i < style="background-color: rgb(228, 228, 228); font-weight: normal; color: rgb(102, 102, 102);" size="11" face="Courier New">"The IP Address is: " + strIpAddress.ToString());

}

catch (SocketException ex)

{

Console.WriteLine("Error Occured! " + ex);

}

Console.Read();

}

}

}

После того как вы запустите вышеприведенную программу, вы получите ip адрес сервера [www.hotmail.com](http://www.hotmail.com/) в нашем случае он отобразился как 211.206.123.219. Однако если вы не подсоединены к сети, метод не сработает и будет получено исключение SocketException. Исключение отлавливается в соответствующем блоке как показано выше в программе. Скриншоты внизу показывают результаты в обоих случаях.

**Создание простого клиент-серверного приложения используя сокеты.**

В этом разделе мы обсудим как создать простое клиент-серверное приложение использую сокеты в C#. У нас будут два различных приложения, приложение Сервер и приложение Клиент. Приложение Сервер будет соединяться с портом и в режиме прослушивания ожидать подключения Клиента. Как только Клиент подсоединиться он пошлет тестовое сообщение Серверу используя StreamWriter. Этот текст затем будет отображен в консоли приложения Сервера.Вот исходный код приложения Сервера.

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

class SocketServer

{

public static void Main()

{

StreamWriter streamWriter;

StreamReader streamReader;

NetworkStream networkStream;

TcpListener tcpListener = new TcpListener(5555);

tcpListener.Start();

Console.WriteLine("The Server has started on port 5555");

Socket serverSocket = tcpListener.AcceptSocket();

try

{

if (serverSocket.Connected)

{

while (true)

{

Console.WriteLine("Client connected");

networkStream = new NetworkStream(serverSocket);

streamWriter = new StreamWriter(networkStream);

streamReader = new StreamReader(networkStream);

Console.WriteLine(streamReader.ReadLine());

}

}

if (serverSocket.Connected)

serverSocket.Close();

Console.Read();

}

catch (SocketException ex)

{

Console.WriteLine(ex);

}

}

}

Относительно фрагмента кода представленного выше. Приложение сервер стартует на порте 5555; отображает соответствующее сообщение и ждет входящего запроса на подсоединение от Клиента. Теперь, когда вы запустили это приложение, сообщение "The Server has started on port 5555", будет отображено на консоли приложения Сервера. Как только Клиент устанавливает соединение на тот же порт, сообщение "Client connected" отображается на консоли. Вот скриншот вывода, после запуска приложения.   
  
  
Далее мы посмотрим на исходный код приложения Клиента, который будет подсоединяться к приложению Сервера используя тот же порт т.е. 5555. Вот код.

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

class SocketClient

{

static void Main(string[]args)

{

TcpClient tcpClient;

NetworkStream networkStream;

StreamReader streamReader;

StreamWriter streamWriter;

try

{

tcpClient = new TcpClient("localhost", 5555);

networkStream = tcpClient.GetStream();

streamReader = new StreamReader(networkStream);

streamWriter = new StreamWriter(networkStream);

streamWriter.WriteLine("Message from the Client...");

streamWriter.Flush();

}

catch (SocketException ex)

{

Console.WriteLine(ex);

}

Console.Read();

}

}

Когда вы выполните приложение Клиента, вот так будет выглядеть вывод консоли приложения Сервера.   
  
Обратите внимание что сообщение "Client connected" и текст посланный клиентом "Message from the Client…", отобразились в консоли приложения Сервера.

**Заключение**

Порт - это беззнаковое целое число которое уникально идентифицирует процесс выполняемый по сети в целях предоставления сервиса. Сокет это конечная точка двухстороннего взаимодействия между двумя процессами выполняемыми в сети. Платформа Microsoft .NET предоставляет отличную поддержку для написания программ которые используют мощь сокетов для воплощения в жизнь программ которые могут работать по сети в целях общения, обмена данными и информацией. В этой статье мы посмотрели на то как мы можем работать с сокетами используя Microsoft .NET и как мы можем реализовать простую клиент-серверную программу которая сможет работать по сети. Комментарии и предложения читателей приветствуются! Приятного чтения!

Опубликовано 11th September 2008 пользователем Denis Kucherov

0

**Добавить комментарий**

**[на тему .net](http://deniskucherov.blogspot.com/" \t "_self)**

**Небольшие заметки и переводы статей на тему разработки в Microsoft .NET Framework**



* [Классический](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=classic)
* [Плитка](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=flipcard)
* [Журнал](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=magazine)
* [Мозаика](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=mosaic)
* [Боковая панель](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=sidebar)
* [Фотоальбом](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=snapshot)
* [Хронология](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html?view=timeslide)

[Feb](http://deniskucherov.blogspot.com/2013/02/nunit-visual-studio-2012.html" \o "7th February 2013)

[7](http://deniskucherov.blogspot.com/2013/02/nunit-visual-studio-2012.html" \o "7th February 2013)

[**NUnit Адаптер в Visual Studio 2012**](http://deniskucherov.blogspot.com/2013/02/nunit-visual-studio-2012.html)

**NUnit Адаптер в Visual Studio 2012**

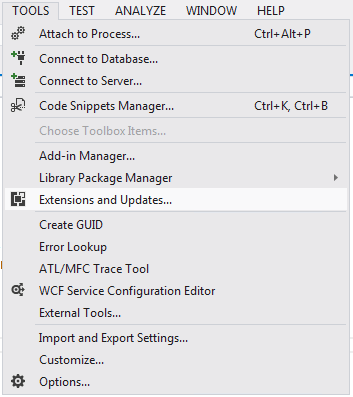
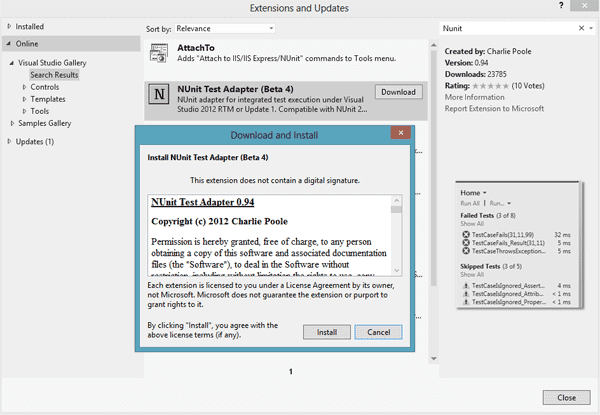
В данной заметки будет коротко изложена процедура установки NUnit под Visual Studio 2012, а та кже приведен пример простого юнит теста в двух вариантах для NUnit и встроенного Ms-Unit.

Популярный фрэймворк для проведения юнит тестирования NUnit теперь работает в Visual Studio 2012. Текущая версия NUnit 0.94 работает в Visual Studio 2012 RTM или с установленным обновлением 1.

**Установка NUnit**

Для установки NUnit Test Adapter можно использовать менеджер расширений который входит в VisualStudio 2012.

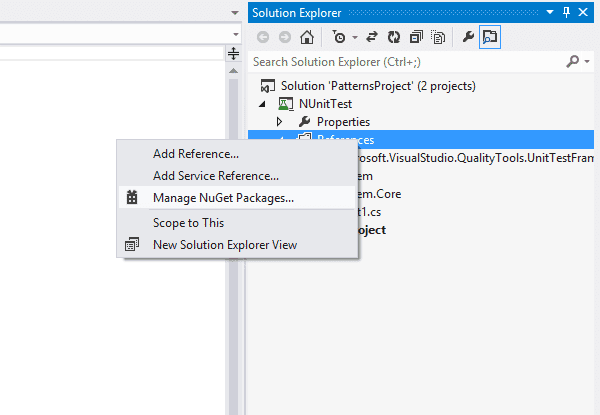
**Шаги установки:**

* 1. В VisualStudio 2012 выберете Инструменты (Tools) | Менеджер расширений (Extension Manager)
  2. [](http://3.bp.blogspot.com/-WDDEg3SvmTM/URNby5jEahI/AAAAAAAAAZM/o4r-i5XO3zw/s1600/UnitTests-Extensions.png)
  3. В левой части менеджера расширений выберете, Онлайн расширения (Online Extensions)
  4. В строке поиска введите NUnit, и выберите в списке NUnit Test Adapter. На момент написания данной статьи версия NUnit 0.94 (Beta 4)
  5. [](http://4.bp.blogspot.com/-AsDd7FxpkH4/URNb4R5xP4I/AAAAAAAAAZg/ljJGQkqhPc4/s1600/UnitTests-Extensions-Install.png)
  6. Нажмите скачать и следуйте инструкции. После установки компонента программа предложит перезагрузить студии для вступления изменений в силу.
  7. [http://3.bp.blogspot.com/-L8kSTi-OENE/URNb4xIGUgI/AAAAAAAAAZs/0y6jPYbM6ro/s1600/UnitTests-restart.png](http://3.bp.blogspot.com/-L8kSTi-OENE/URNb4xIGUgI/AAAAAAAAAZs/0y6jPYbM6ro/s1600/UnitTests-restart.png)

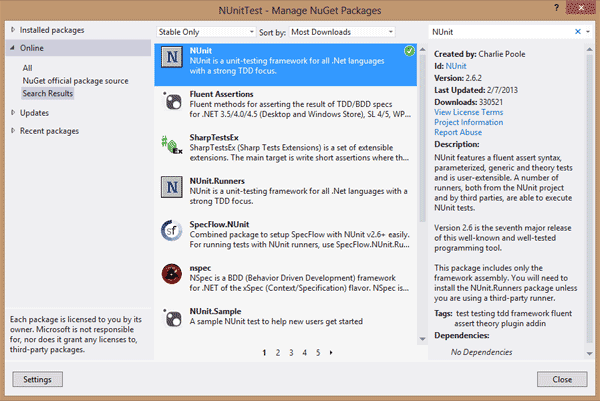
**Использование NUnit**

Для начала использования NUnit убедитесь, что у вас открыто окно Тестирования. Для этого перейдите в раздел Тест | Окна и установите галочку напротив Проводник Тестов (Test Explorer).

Добавьте новый тестовый проект. Теперь необходимо установить библиотеку тестирования NUnit для этого перейдите в вновь созданный тестовый проект и зайдите в менеджер NUGet.

[](http://4.bp.blogspot.com/-yNbLamvamRM/URNb4LkrMXI/AAAAAAAAAZU/2OW91wbv7VE/s1600/Uni-Tests-NuGet.png)

В открывшемся окне менеджера NUGet наберите в строке поиска NUnit и установите найденный дистрибутив библиотеки.

[](http://4.bp.blogspot.com/-cqQBcc1b18w/URNb4hO1PmI/AAAAAAAAAZw/5EdDTRFVS68/s1600/UnitTests-NuGet-install.png)

**Пример теста NUnit vs MS-Unit**

В приведенном примере я сделал простой тест для объекта заказа. Данный заказ является частью демо проекта в котором реализован паттерн Strategy (стратегия). Тест создает объект заказа и передает в конструкторе два параметра которые должны быть присвоены полям объекта.

На скриншоте изображен тест и результат работы в левом окне. Обратите внимание что ссылка на пространство имен Microsoft.VisualStudio.TestTools.UnitTesting не требуется. Библиотеку Microsoft.VisualStudio.QualityTools.UnitTestFramework из проекта можно удалить.

Пример теста запущенного под управлением встроенной библиотеки ms-Unit.

 Обратите внимание на время выполнения. NUnit думает подольше при первом запуске теста.

**Вывод**

В Visual Studio позволяет использовать сторонние фрэймворки юнит тестирования, такие как NUnit, MbUnit и xUnit.Net. При это результаты работы пользователь получает в стандартном окне студии.

**Источники:**

<http://www.codeguru.com/csharp/.net/using-nunit-with-visual-studio-2012-unit-test-projects.htm>

<http://nunit.org/index.php?p=vsTestAdapter&r=2.6.2>

Опубликовано 7th February 2013 пользователем Denis Kucherov

0

**Добавить комментарий**

[Apr](http://deniskucherov.blogspot.com/2011/04/solid.html" \o "22nd April 2011)

[22](http://deniskucherov.blogspot.com/2011/04/solid.html" \o "22nd April 2011)

[**SOLID**](http://deniskucherov.blogspot.com/2011/04/solid.html)

пять основных принципов дизайна классов в ООП

**SRP - The Single Responsibility Principle**

class should have one, and only one, reason to change.  
Принцип единственной обязанности  
на каждый объект должна быть возложена одна единственная обязанность.  
  
[Подробное описания принципа](http://www.objectmentor.com/resources/articles/srp.pdf) (ENG) (pdf)

**OCP - The Open Closed Principle**

You should be able to extend a classes behavior, without modifying it.  
Принцип открытости/закрытости  
Программные сущности должны быть открыты для расширения, но закрыты для изменения.  
  
[Подробное описания принципа](http://www.objectmentor.com/resources/articles/ocp.pdf) (ENG) (pdf)

**LSP - The Liskov Substitution Principle**

Derived classes must be substitutable for their base classes.  
Принцип подстановки Лисков  
Объекты в программе могут быть заменены их наследниками без изменения свойств программы.   
  
[Подробное описания принципа](http://www.objectmentor.com/resources/articles/lsp.pdf) (ENG) (pdf)

**ISP - The Interface Segregation Principle**

Make fine grained interfaces that are client specific.  
Принцип изоляции интерфейса  
Много специализированных интерфейсов лучше, чем один универсальный.  
  
[Подробное описания принципа](http://www.objectmentor.com/resources/articles/isp.pdf) (ENG) (pdf)

**DIP - The Dependency Inversion Principle**

Depend on abstractions, not on concretions.  
  
Принцип инверсии зависимостей  
Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций  
  
[Подробное описания принципа](http://www.objectmentor.com/resources/articles/dip.pdf) (ENG) (pdf)   
  
  
Источники:   
[http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod](https://www.blogger.com/butunclebob.com)  
[ru.wikipedia.org](http://ru.wikipedia.org/wiki/SOLID_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))

Опубликовано 22nd April 2011 пользователем Denis Kucherov

Ярлыки: [Dev](http://deniskucherov.blogspot.com/search/label/Dev)

0

**Добавить комментарий**

[Apr](http://deniskucherov.blogspot.com/2011/04/mix11.html" \o "19th April 2011)

[19](http://deniskucherov.blogspot.com/2011/04/mix11.html" \o "19th April 2011)

[**Опубликованы материалы MIX'11**](http://deniskucherov.blogspot.com/2011/04/mix11.html)

Опубликованы материалы и записи прошедшей конференции MIX’11.   
В ходе конференции было представлено большое количество докладов на темы разработки и дизайна ПО.  
  
Посмотреть технические доклады можно на сайте   
[live.visitmix.com/](http://live.visitmix.com/).  
  
В этой записи блога Владимира Юнеева удобно представлен перечень докладов по технологиям.   
[blogs.msdn.com/b/vyunev/archive/2011/04/11/mix-11.aspx](http://blogs.msdn.com/b/vyunev/archive/2011/04/11/mix-11.aspx)

Опубликовано 19th April 2011 пользователем Denis Kucherov

Ярлыки: [Microsoft](http://deniskucherov.blogspot.com/search/label/Microsoft)

0

**Добавить комментарий**

[Sep](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html" \o "11th September 2008)

[11](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html" \o "11th September 2008)

[**Программирование сокетов в C#**](http://deniskucherov.blogspot.com/2009/09/by-joydip-kanjilal-httpaspalliance.html)

by [Joydip Kanjilal](http://aspalliance.com/author.aspx?uId=58780)   
оригинал статьи: <http://aspalliance.com/1563_Socket_Programming_in_C.all>   
перевод Denis M Kucherov

**Краткое изложение**

В этой статье Joydip продемонстрирует работу сокетов при программировании в C#. Он начнет с краткого ознакомления с сокетами и портами рассмотрит разницу между синхронной и асинхронной передачей данных. Вы узнаете как работать с пространствами имен System.Net и System.Net.Sockets применяя их в простых клиент-серверных приложениях. Статья также снабжена всеми необходимыми примерами кода и результатами их работы в виде скрин-шотов.

**Вступление**

Компьютерная сеть предполагает наличие двух или более компьютеров соединенных вместе посредствам оборудования и программного обеспечения, в целях обмена данными и информацией. Microsoft .NET предоставляет отличную поддержку в написании программ которые могут использовать сокеты для обмена данными между двумя и более компьютерами в сети. По мере чтения данной статьи мы будем узнавать больше о сокетах и портах. В данной статье расматривается то , как мы можем работать с сокетами в .NET, в целях написания программ которые могут работать в сети. Обратите внимание на то что примеры приведенные в статье написаны на C#.

**Понимание Сокетов и Портов**

Сокет, это конечная точка двухстороннего соединения между двумя системами работающих в сети. Когда два или более процесса взаимодействуют через сеть, они взаимодействуют используя сокеты. Порт представляет собой беззнаковое целое число, которое уникально идентифицирует процесс выполняющийся в сети.   
Вот некоторые хорошо известные номера портов охарактеризованные типами сервисов которые их предоставляют:

* 1. HTTP 80
  2. Telnet 23
  3. SMTP 25
  4. SNPP 444
  5. DNS 53
  6. FTP (Data) 20
  7. FTP (Control) 21

Синхронное и асинхронное соединение   
В Microsoft .NET вы имеете поддержку обоих, синхронного и асинхронного соединения при использовании сокетов. Также известный как, блокированный и не блокированный режим работы. Существуют весьма тонкие различия между двумя этими методами. Когда работаем в синхронном режиме, вызов метода блокирует сам себя до тех пор пока операция не будет полностью закончена. В другом режиме работы, т.е. асинхронном режиме, метод возвращается еще до того как время цикла обработки закончилось. Отлично, но что такое время цикла обработки? Под временем цикла обработки подразумевается общее время затраченное потоком до полного его завершения.   
В синхронном режиме взаимодействия, серверное приложение слушает конкретный порт на предмет получения данных от клиента. Поступая так, серверное приложение блокируется (для других клиентов) до тех пор пока не получит данные от клиентского приложения. С другой стороны, во время работы в асинхронном режиме, сервер может обрабатывать множество клиентских запросов одновременно. Обратите внимание, что асинхронные команды использующие сокеты обычно применяются для задач которые требуют больших затрат времени. Типичные примерами таких задач является открытие больших файлов, отправка запросов базе данных с большим объемом данных, подсоединение к удаленному компьютеру, удаленный доступ к ресурсам требующих больших временных затрат.   
Так же учтите, что асинхронные вызовы на самом деле работают в раздельных потоках.   
Обычно программа имеет два вида потоков, программный поток и рабочий поток. Программный поток - это основной поток программы; рабочий поток - это поток работающий в фоне, для обеспечения асинхронных операций.   
Класс Socket в пространстве имен System.Net.Sockets содержит и синхронные и асинхронные методы. К примеру, в то время как Connect() и Recive() методы предназначены для синхронного использования, BeginConnect() и EndConnect() методы, так же как BeginReceive() и EndReceive() методы их асинхронные аналоги.

**Работа с пространствами имен System.Net и System.Net.Sockets**

Класс Socket в пространстве имен System.Net.Sockets используется для работы с сокетами в C#. Заметьте что с каждым сокетом ассоциируется локальные и удаленные конечные точки. Мы рассмотрим этот класс позднее в статье.   
Пространство имен System.Net включает в себя важный клас под названием Dns, который может быть использован для доступа к DNS (Domain Naming Service). Так что же такое DNS? Domain Naiming Service или DNS это сервис имен который используется для именования узлов сети с целью упрощения. Имя домена это текстовое имя которое идентифицирует главный компьютер (host) в сети. DNS сервера хранят DNS адреса совместно с соответствующими им IP адресами.   
Теперь, для того что бы получить хост-имя вашего локального компьютера, используйте следующее:

Console.WriteLine("The host name of the local computer is: " + Dns.GetHostName());

Хост-имя локального компьютера должно отобразиться, как показано ниже.

На верхнем фрагменте кода, GetHostName() статический метод класса Dns. Вот пример программы которая илюстрирует то, как вы можите отобразить IP адрес сайта [www.hotmail.com](http://www.hotmail.com/) используя этот класс.

**Отображение IP адресов сервера Hotmail используя сокеты**

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace TestSocket

{

class Program

{

static void Main(string[]args)

{

try

{

IPHostEntry IPHost = Dns.GetHostEntry("www.hotmail.com");

IPAddress[]ipAddress = IPHost.AddressList;

StringBuilder strIpAddress = new StringBuilder();

for (int i = 0; i < style="background-color: rgb(228, 228, 228); font-weight: normal; color: rgb(102, 102, 102);" size="11" face="Courier New">"The IP Address is: " + strIpAddress.ToString());

}

catch (SocketException ex)

{

Console.WriteLine("Error Occured! " + ex);

}

Console.Read();

}

}

}

После того как вы запустите вышеприведенную программу, вы получите ip адрес сервера [www.hotmail.com](http://www.hotmail.com/) в нашем случае он отобразился как 211.206.123.219. Однако если вы не подсоединены к сети, метод не сработает и будет получено исключение SocketException. Исключение отлавливается в соответствующем блоке как показано выше в программе. Скриншоты внизу показывают результаты в обоих случаях.

**Создание простого клиент-серверного приложения используя сокеты.**

В этом разделе мы обсудим как создать простое клиент-серверное приложение использую сокеты в C#. У нас будут два различных приложения, приложение Сервер и приложение Клиент. Приложение Сервер будет соединяться с портом и в режиме прослушивания ожидать подключения Клиента. Как только Клиент подсоединиться он пошлет тестовое сообщение Серверу используя StreamWriter. Этот текст затем будет отображен в консоли приложения Сервера.Вот исходный код приложения Сервера.

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

class SocketServer

{

public static void Main()

{

StreamWriter streamWriter;

StreamReader streamReader;

NetworkStream networkStream;

TcpListener tcpListener = new TcpListener(5555);

tcpListener.Start();

Console.WriteLine("The Server has started on port 5555");

Socket serverSocket = tcpListener.AcceptSocket();

try

{

if (serverSocket.Connected)

{

while (true)

{

Console.WriteLine("Client connected");

networkStream = new NetworkStream(serverSocket);

streamWriter = new StreamWriter(networkStream);

streamReader = new StreamReader(networkStream);

Console.WriteLine(streamReader.ReadLine());

}

}

if (serverSocket.Connected)

serverSocket.Close();

Console.Read();

}

catch (SocketException ex)

{

Console.WriteLine(ex);

}

}

}

Относительно фрагмента кода представленного выше. Приложение сервер стартует на порте 5555; отображает соответствующее сообщение и ждет входящего запроса на подсоединение от Клиента. Теперь, когда вы запустили это приложение, сообщение "The Server has started on port 5555", будет отображено на консоли приложения Сервера. Как только Клиент устанавливает соединение на тот же порт, сообщение "Client connected" отображается на консоли. Вот скриншот вывода, после запуска приложения.   
  
  
Далее мы посмотрим на исходный код приложения Клиента, который будет подсоединяться к приложению Сервера используя тот же порт т.е. 5555. Вот код.

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

class SocketClient

{

static void Main(string[]args)

{

TcpClient tcpClient;

NetworkStream networkStream;

StreamReader streamReader;

StreamWriter streamWriter;

try

{

tcpClient = new TcpClient("localhost", 5555);

networkStream = tcpClient.GetStream();

streamReader = new StreamReader(networkStream);

streamWriter = new StreamWriter(networkStream);

streamWriter.WriteLine("Message from the Client...");

streamWriter.Flush();

}

catch (SocketException ex)

{

Console.WriteLine(ex);

}

Console.Read();

}

}

Когда вы выполните приложение Клиента, вот так будет выглядеть вывод консоли приложения Сервера.   
  
Обратите внимание что сообщение "Client connected" и текст посланный клиентом "Message from the Client…", отобразились в консоли приложения Сервера.

**Заключение**

Порт - это беззнаковое целое число которое уникально идентифицирует процесс выполняемый по сети в целях предоставления сервиса. Сокет это конечная точка двухстороннего взаимодействия между двумя процессами выполняемыми в сети. Платформа Microsoft .NET предоставляет отличную поддержку для написания программ которые используют мощь сокетов для воплощения в жизнь программ которые могут работать по сети в целях общения, обмена данными и информацией. В этой статье мы посмотрели на то как мы можем работать с сокетами используя Microsoft .NET и как мы можем реализовать простую клиент-серверную программу которая сможет работать по сети. Комментарии и предложения читателей приветствуются! Приятного чтения!

Опубликовано 11th September 2008 пользователем Denis Kucherov

0

**Добавить комментарий**

[Apr](http://deniskucherov.blogspot.com/2009/09/joe-levi-weblogs.html" \o "13th April 2008)

[13](http://deniskucherov.blogspot.com/2009/09/joe-levi-weblogs.html" \o "13th April 2008)

[**ASP.NET DataReader vs. DataSet**](http://deniskucherov.blogspot.com/2009/09/joe-levi-weblogs.html)

Автор: Joe Levi  
Оригинал: weblogs.asp.net/joelevi/archive/2008/02/12/asp-net-datareader-vs-dataset.aspx  
Перевод: Denis Kucherov  
  
Недавно я заново попробовал "использовать правильные инструменты для рабы" при разработке xhtml и CSS, что заставило меня задуматься. Если использование 'ul' вместо 'div' предпочтительнее в определенных программах, есть ли инструменты в ASP.NET использование которых для некоторых задач более предпочтительно. Вот с такой мыслью я взялся за задачу которая требует соединения с одной из наших баз данных. Вместо построения DAL (я пока еще не очень с LINQ) я решил просто вытащить данные через SqlConnection и дальше работать с ними. С этим все понятно. Но затем я столкнулся с тем, как "работать" с данными полученными от SqlCommand. В .NET есть несколько контролов которые умеют читать данные из SqlCommand: DataReader и DataSet. Но какой использовать?  
  
Я всегда старался использовать то, в чем хорошо разбираюсь, и понимать при этом полученный результат. Практически во всех моих случаях я использовал DataSet контрол. Это позволяло мне не только получать данные из базы, но так же обновлять и вставлять данные. В этом смысле DataSet идеален. Но со всей этой дополнительной функциональностью, приходит дополнительная нагрузка.  
  
С другой стороны DataReader "не делает все за вас". Он делает свою работу, и делает ее хорошо, и более ничего (звучит как концепция UNIX, не так ли?) . С DataReader вы можите только читать данные из базы, и управлять ею (преобразовать в объект, вывести на страницу, и тп...) и закрыть SqlConnection.  
  
4GuysFromRolla так описали DataReader:  
"[...]DataReader всего лишь передача данных между программой и базой данных.  
[...] к тому же возможности DataReader-а лимитированы только чтением данных в прямолинейном направлении (только вперед, последовательно).  
Тоесть информация полученная из базы не может быть модифицирована DataReader-ом, и так же DataReader не может получать данные в произвольном порядке. DataReader ограничен доступом к данным в последовательном порядке, от начала до конца по одной записи за раз."  
  
Вау, походит на ужасные ограничения, не так ли? Если вы хотите делать обновление и вставку данных, тогда конечно. Если же вы только получаете данные из базы, DataReader более всего подходит на роль идеального инструмента для этой работы: Открываем соединение получаем данные, обрабатываем и закрываем соединение. И это то где DataReader действительно блистает; Взгляните на график.  
  
  
По всему движению графика время не сильно увеличивается (синяя линия). А вот вторая линия (красная) даже при возврате 100 записей результат тревожный, более 1000 записей и вы легко можите увидеть почему предпочтете использовать DataReader вместо DataSet (опять же за исключением случаев когда вам нужны дополнительные возможности которые предоставляет DataSet).  
  
Вы можите прочитать больше (и изучить все детали результатов теста) на 4GuysFromRolla.

Опубликовано 13th April 2008 пользователем Denis Kucherov

0

**Добавить комментарий**

[Feb](http://deniskucherov.blogspot.com/2008/02/aspnet-mvc-framework-1.html" \o "16th February 2008)

[16](http://deniskucherov.blogspot.com/2008/02/aspnet-mvc-framework-1.html" \o "16th February 2008)

[**ASP.NET MVC Framework (Часть 1)**](http://deniskucherov.blogspot.com/2008/02/aspnet-mvc-framework-1.html)

Автор: [ScottGu](http://weblogs.asp.net/members/ScottGu.aspx)  
Оригинал статьи: <http://weblogs.asp.net/scottgu/archive/2007/11/13/asp-net-mvc-framework-part-1.aspx>  
Перевод: Denis Kucherov  
  
Две недели назад (Ноябрь 2007) я писал в своем блоге о новом паттерне [MVC (Model View Controller)](http://weblogs.asp.net/scottgu/archive/2007/10/14/asp-net-mvc-framework.aspx) для ASP.NET который мы будем поддерживать как дополнительную возможность. Паттерн предоставляет структурированную модель которая дает возможность полного разделения задач внутри приложения, упрощает тестирование модулей и организацию рабочего процесса с применением тестов ([TDD](http://www.citforum.ru/SE/testing/mod_test/) программирование на основе тестов). Он также дает больший контроль над ссылками (URL) которые вы используете в приложении, и при желании может предоставлять больший контроль над HTML страницами на которые эти ссылки указывают.  
  
  
С тех пор я постоянно отвечаю на массу вопросов людей которые жаждут узнать как можно больше на эту тему. Учитывая уровень интереса к теме я решил что имеет смысл написать несколько статей с более детальным описание использования этой техники. Эта статья первая из нескольких которые я планирую написать в ближайшую неделю.

**Простая программа магазина электронной коммерции.**

Я планирую использовать простое приложение интернет магазина которое поможет проиллюстрировать как ASP.NET MVC Framework работает. Для целей сегодняшней статьи я применю сценарий вывода/просмотра продукции.  
По сути, мы сделаем прилавок магазина который дает возможность конечному пользователю просматривать список категорий продукции когда они посещают адрес *Products/Categories на сайте:*   
  
Когда пользователь кликнет на гиперссылку группы товара (верхний рисунок), они перейдут на список этой категории */Products/List/CategoryName который выводит список продуктов выбранной категории:*   
Когда пользователь кликнет на конкретный продукт, он перейдет по адресу подробной информации о продукте *URL - /Products/Detail/ProductID* - который отображает более детальную информацию о выбранном продукте.   
Мы построим весь этот функционал используя новый каркас ASP.NET MVC. Это позволит нам добиться полного разделения ролей среди разных компонентов приложения, и даст возможность легко встроит модульное тестирование и использовать программирование на основе тестов.

**Создание новой ASP.NET MVC программы**

Каркас ASP.NET MVC содержит шаблоны проектов Visual Studio, что облегчает процесс создания нового веб приложения. Чтобы создать сайт просто зайдите в File->New Project меню и выберите шаблон "ASP.NET MVC Web Application".  
По умолчанию, когда вы создаете новое веб приложение используя этот шаблон, Visual Studio создаст для вас новое решение и добавит в него два проекта. Первый проект - это веб проект в которым вы будите создавать свое приложение. Второй - это тестовый проект который вы можите использовать для написания тестов для веб проекта:   
  
С ASP.NET MVC Framework вы можите использовать любой каркас модульного тестирования включая (NUnit, MBUnit, MSTest, XUnit и другие). VS 2008 Professional теперь включает в себя встроенный тестовый проект с поддержкой MSTest (в прошлом в VS 2005 это требовало версии Visual Studio Team System SKU), и наш стандартный ASP.NET MVC шаблон проекта, автоматически создает один из таких проектов когда вы используете VS 2008.  
Также мы сделали доступные для скачивания шаблоны проектов для NUnit, MBUnit а так же других каркасов модульных тестов. Так что если вы предпочитаете использовать их, у вас так же есть простая возможность создать проект одним нажатием с уже готовым для работы проектом для тестирования.

**Понимание структуры папок проекта**

Структура программы ASP.NET MVC имеет 3 корневые папки:

* 1. /Controllers
  2. /Models
  3. /Views

Как вы видимо уже догадались, мы рекомендуем вам размещать классы Контроллера в папке /Controller, классы модели данных в папке /Models, а шаблоны представления в папке /Views.  
ASP.NET MVC framework не обязывает вас всегда придерживаться этой структуры, однако шаблоны проектов используют по умолчанию такую схему и мы рекомендуем использовать ее как удобный способ структурирования вашей программы. Если у вас нет веской причины использовать другое расположение файлов, я рекомендую пользоваться схемой по умолчанию.

**Передача (проецирование) URL в классы Контроллеров**

В большинстве Веб движков (ASP, PHP, JSP, ASP.NET WebForms, и тд.) входящие URL просто указывают на файлы хранящиеся на диске. К примеру "/Products.aspx" или "/Products.php" URL имеет в своей основе находящийся на диске фал Products.aspx или Products.php который управляет его обработкой. Когда http запрос для веб программы поступает на веб сервер, ядро веб сервера запускает код находящийся внутри нужного файла, и этот код самостоятельно управляет обработкой запроса. Зачастую содержание этих файлов Products.aspx или Products.php включает HTML код который помогает сформировать ответ отсылаемый назад клиенту.  
MVC Framevork подругому увязывает URL адреса с серверным кодом. Вместо того что бы связывать URL с определенным файлом на диске, MVC связывает URL напрямую с классами. Эти классы называются "Контроллеры" и они самостоятельно обрабатывают входящие запросы, управляют пользовательскими параметрами и обменом данных, выполняют необходимые операции и бизнес логику основанную на них. Класс Контроллера затем вызывает отдельный компонент Представления (View), который самостоятельной формирует необходимый HTML код для возврата в качестве ответа.   
ASP.NET MVC Framework имеет очень мощный движок URL проецирования, который имеет достаточно гибкие возможности по передачи URL в класс Контроллера. Вы можите использовать его для легкой установки правил распределения, которые ASP.NET затем будет использовать для оценки поступившего URL и выбора Контроллера для выполнения.  
Движок распределения также автоматически преобразует переменные определенные в URL а ASP.NET передает их в ваш Контроллер в виде аргументов параметров. Я разберу более сложные примеры включающие движок URL распределения в следующий статьях этой серии.

**Принятое по умолчанию в ASP.NET MVC URL распределение по классам Контроллеров**

По умолчанию ASP.NET MVC проект имеет предустановленные настройки правил URL распределения, что позволяет вам легко запустить приложение без необходимости что либо конфигурировать явным образом. Вместо этого вы можите начать программировать используя настройки по умолчанию в основе которых, основанное на именах URL распределение, прописанное в специальном формате в ASP.NET в классе Application файла Global.asax созданного новым шаблоном проекта ASP.NET MVC.   
Основное соглашение по именам состоит в том что бы проецировать начальный URL путь входящего HTTP запроса (например: /Products/) на класс, имя которого отвечает шаблону *UlrPath*Controller (например: по умолчанию URL начинающийся на /Products/ будет спроектирован на класс с именем *Products*Controller).  
Для построения функционала просмотра нашего коммерческого продукта, мы добавим в наш проект новый класс "ProductController" (вы можите использовать "Add New Item" меню в Visual Studio) для легкого создания класса Контроллера из шаблона.   
Наш класс ProductsController будет наследоваться от базового класса System.Web.MVC.Controller. Наследование от этого базового класса не обязательное, однако он (базовый класс) включает некоторые полезные методы и функционал от которого в дальнейшем мы планируем получить пользу:   
Как только мы определили класс ProductsController в нашем проекте, ASP.NET MVC framework будет по умолчанию использовать его для обработки всех входящих URL которые подчинены пространству имен "/Products/". Это значит он будет автоматически вызван для обработки URL-ов "/Products/Categories", "/Products/List/Beverages", и"/Products/Detail/3", которые мы собираемся использовать применительно к нашему приложению интернет-прилавка.   
В следующей статье мы также добавим ShoppingCartController (что бы дать пользователям управлять их корзиной покупок) и AccountController (что бы дать возможность пользователям создавать новые учетные записи на сайте и осуществлять вход/выход). После того как мы добавим эти два новых класса контроллеров в наш проект, URL-ы которые начинаются с /ShoppingCart/ and /Account/ будут автоматически перенаправляться им для обработки.   
*Примечание: ASP.NET MVC framework не заставляет вас постоянно придерживаться такому шаблону правил именования. Единственная причина по которой наша программа использует этот шаблон по умолчанию состоит в том, что присутствуют правила распределения которые сконфигурировали это, которые в свою очередь были автоматически добавлены в наш ASP.NET Application класс, когда мы создали новый ASP.NET MVC проект используя Visual Studio.*

**Понимание методов Контроллера**

Теперь когда мы создали класс ProductsController в нашем проекте, мы можем начать добавлять логику управления обработкой URL-а "/Products/" поступающего в наше приложение.  
Когда в начале этой статьи определялся общий вид нашего интернет прилавка, я сказал, что мы будем реализовывать 3 сценария для этого сайта: 1) Просмотр всех категорий продуктов, 2) Просмотр продуктов в конкретной категории, 3) Просмотр подробной информации по отдельному продукту. Мы будем использовать, указанные ниже, оптимизированные для поисковых систем (SEO) адреса URL, для обработки каждого из этих сценариев:

|  |  |  |
| --- | --- | --- |
| **Формат URL** | **Действие** | **Пример URL** |
| /Products/Categories | Показывает все категории продуктов | /Products/Categories |
| /Products/List/Category | Показывает продукты в категории | /Products/List/Beverages |
| /Products/Detail/ProductID | Детальная информация о конкретном продукте | /Products/Detail/34 |

Существует несколько способов того, как мы можем написать код в ProductsController классе для обработки этих трех типов входящих URL. Одним из способов было бы перегрузить метод "Execute" базового класса Контроллера и написать наш собственный if/else/switching блок для просмотра входящего URL и затем выполнять необходимую логику.   
Думаю более легкий подход использовать встроенную возможность каркаса MVC, которая позволяет нам определить "методы действия" в нашем контроллере, и затем базовый класс Контроллера сам применит нужный метод основываясь на правилах URL распределения используемых в нашем приложении.   
Например мы можем дабавить как показано ниже три метода Контроллера в наш класс ProductController, для осуществления наших 3-х URL сценариев электронной-комерции рассмотренных выше:   
Правила URL распределения сконфигурированные по умолчанию когда был создан проект, рассматривают URL подкаталог который идет после имени Котроллера как имя действия запроса. Таким образом если мы получаем URL запрос вида "/Products/Categories", правило распределения расценит "Categories" как имя действия и метод Categories() будет запущен для обработки запроса. Если мы получим URL запрос вида "/Products/Detail/5", правило распределения расценит "Detail" как имя действия, и метод Detail() будет выполнен для обработки запроса, и т.п...   
*Примечание: ASP.NET MVC framework не обязывает вас всегда использовать такой шаблон взаимосвязи имен. Если вы хотите использовать другой шаблон наложения URL, просто пойдите в ASP.NET Application Class (в Global.asax) и измените его.*

**Передача URL параметров в исполняемые методы Контроллера**

Существует несколько способов передачи значений параметров URL в исполняемые методы класса Контроллера. Базовый класс контроллера представляет пару Request и Response объектов которые могут быть использованы. Эти объекты имеют абсолютно такую же API структуру как объекты HttpRequest/HttpResponse с которыми вы уже знакомы из ASP.NET. Одно важное отличие состоит в том что эти объекты теперь основаны на интерфейсе взамен изолированного (sealed) класса (Замечание: MVC framework идет с новыми интерфейсами System.Web.IHttpRequest и System.Web.IHttpResponse). Преимущество использования этих объектов как интерфейсов состоит в том, что теперь проще их имитировать - что позволяет легко производить модульное тестирование (UnitTesting) классов контроллера. Я рассмотрю эту тему более подробно в следующих статьях.   
Ниже приведен пример того как мы можем использовать Request API для самостоятельного извлечения ID из строки запроса, в нашем исполняемом методе Detail() , в классе ProductController:   
ASP.NET MVC framework так же поддерживает автоматическое преобразование входящих URL значений параметров, как аргументов параметров в метод выполнения. По умолчанию, если у вашего метода выполнения есть аргумент параметра, MVC framework "посмотрит" на входящие данные строки запроса, есть ли там соответствующие HTTP значения с таким же именем. Если есть, MVC framework автоматически отправит их в качестве параметра в метод исполнения.   
Для примера мы можем переписать наш Detail() метод исполнения, чтобы воспользоваться этой возможностью и сделать метод более "чистым" как показано ниже:   
В добавок возможности преобразования значений аргументов из коллекций строки/формы запроса, ASP.NET MVC framework так же позволяет вам использовать инфраструктуру MVC URL распределения маршрутов для встраивания значений параметров во внутрь URL (например: вместо "/Products/Detail?id=3" вы можите использовать "/Products/Detail/3").   
Правила распределения маршрутов задаваемые в момент создания нового MVC проекта, имеют такой формат: "/[controller]/[action]/[id]". Это значит что если в адресе URL имеется какой либо URL подкаталог после контроллера и имени действия, то он по умолчанию будет рассматриваться как имя параметра "id" - который может быть в свою очередь передан в наш метод контроллера как аргумент.   
Это значит что мы теперь так же можем использовать наш метод Detail для взятие аргумента ID из URL пути (например: "/Products/Detail/3"):   
Я могу использовать подобный подход для действия List и тогда мы сможем послать наименование категории, как часть URL (например: "/Products/List/Beverages"). Для того чтобы сделать этот код боле читабельным, я проделал одну хитрость в правилах распределения, сделал так что вместо использования аргумента под названием "id" для этого действия будет вызван "category".   
Ниже показана версия нашего класса ProductsController в котором полностью применено URL распределение и конвертация параметров:   
Обратите внимание на то что метод List принимает параметр категории как часть URL, и затем не обязательный параметр индекса страницы как строки запроса (мы применим разбиение страниц на стороне сервера и будем использовать это значение для определения того какую страницу категорий данных отображать).   
В нашей программе не обязательный параметр обрабатывается используя nullable тип аргумента в методе Action. Так как параметр page в нашем методе List имеет тип nullable int (имено это означает "int?"), MVC framework либо передаст его в качестве значения представленного в URL, либо в качестве null если в URL нет значения. Вы можите почитать мою [предыдущую статью о специальном операторе ??](http://weblogs.asp.net/scottgu/archive/2007/09/20/the-new-c-null-coalescing-operator-and-using-it-with-linq.aspx) из которой можно узнать о полезных способах и хитростях того как работать с nullable типами.

**Построение нашего объекта модели данных**

Теперь у нас есть класс ProductsController и три метода действия в нем готовых обрабатывать входящие Веб запросы. Нашим следующим шагом будет построение некоего класса помогающего нам работать с базой данных для получения необходимой информации для обработки этих веб запросов.   
В мире MVC "модели" это компоненты приложения которые ответственны за поддержание структуры. Касательно веб-приложения эта структура обычно представлена в базе данных (например: у нас может быть объект Product, который используется для представления информации о продуктах из таблицы Products внутри нашей SQL базы данных).   
ASP.NET MVC Framework позволяет использовать любые на ваш выбор шаблоны или объекты доступа к информации в целях получения и управления данными. Если вы хотите вы можите использовать ADO.NET DataSets/DataReaders (или абстрактный слой поверх их). Если вы предпочитаете использовать объектно-реляционное отображение (ORM) типа NHibernate, LLBLGen, WilsonORMapper, LINQ to SQL/LINQ to Entities, все это вы можете так же использовать.   
Для нашего примера приложения электронной коммерции я буду использовать встроенную LINQ to SQL ORM поставку в NET 3.5 and VS 2008. Вы можите узнать больше информации о LINQ to SQL из [моих серий инструкций](http://weblogs.asp.net/scottgu/archive/2007/09/07/linq-to-sql-part-9-using-a-custom-linq-expression-with-the-lt-asp-linqdatasource-gt-control.aspx) на эту тему (обязательно посмотрите [Часть1](http://weblogs.asp.net/scottgu/archive/2007/05/19/using-linq-to-sql-part-1.aspx), [Часть2](http://weblogs.asp.net/scottgu/archive/2007/05/29/linq-to-sql-part-2-defining-our-data-model-classes.aspx), [Часть3](http://weblogs.asp.net/scottgu/archive/2007/06/29/linq-to-sql-part-3-querying-our-database.aspx) и [Часть4](http://weblogs.asp.net/scottgu/archive/2007/07/11/linq-to-sql-part-4-updating-our-database.aspx) статей).   
Я начну правым кликом на подкаталог "Models" нашего MVC веб проекта внутри VS и выберу "Add New Item" для добовления LINQ to SQL шаблона. С LINQ to SQL ORM дизайнером я создам три класса модели данных которые составляют схему для таблиц Categories, Products, и Suppliers входящих в демонстрационную базу SQL Server Northwind (читайте [Часть2](http://weblogs.asp.net/scottgu/archive/2007/05/29/linq-to-sql-part-2-defining-our-data-model-classes.aspx) моей серии LINQ to SQL чтобы узнать как это сделать):   
  
Как только мы задали наш LINQ to SQL класс модели данных, я так же добавил новый NorthwindDataContext парциальный (partial) класс в нашу папку Models:   
Внутри этого класса я определю несколько вспомогательных методов, которые инкапсулируют некоторые LINQ выражения. Их мы сможем использовать для получения уникальных объектов Category из нашей базы данных, получения всех объектов Product входящих в конкретную категорию в нашей базе, а так же для получения индивидуального объекта Product основываясь на предоставленном идентификаторе:   
Эти вспомогательные методы сделают для нас боле простым получение объектов модели данных необходимых для нашего ProductsController класса (без обходимости писать LINQ выражения в самом классе контроллера):   
Теперь у нас есть весь необходимый код/объекты данных, для того что бы закончить реализацию нашего ProductsController функционала.

**Заканчиваем реализацию нашего ProductsController класса**

Контроллеры в программах основанных на MVC ответственны за обработку входящих запросов, управляют пользовательским вводом и взаимодействием, и выполняют основанную на них соответствующую программную логику (получение и обновление объекта данных хранящегося в базе данных и т.п. ).   
Контроллеры обычно не создают нужный HTML ответ для запроса. Задача создания HTML ответа принадлежит компонентам "View"реализованного в рамках приложения в виде отдельных от контроллеров классов/шаблонов. Представления (Views) предназначены для того, что бы полностью сфокусироваться на инкапсуляции логики отображения, и не должно включать ни какой логики приложения, или кода запроса к базе данных (вместо этого вся программная логика должна быть в Контроллере).   
В типичном MVC рабочем потоке веб приложения, методы действия Контроллера будут управлять входящими веб запросами, использовать значения входящих параметров для выполнения соответствующего кода программной логики, получать или обновлять объект модели данных из базы данных, и затем выбирать "Представление" в целях выдачи необходимого UI (интерфейс пользователя) ответа назад в браузер. Вместе с выбором необходимого Представления для вывода, Контроллер так же явным образом передаст (в качестве аргументов) все данные и переменные необходимые Представлению в целях вывода соответственного ответа:   
Вы возможно хотите спросить - в чем же приимущество разделения Контроллера и Представления таким способом? Почему бы просто не поместить их в один класс? Основная цель разделения приложения таким образом в том что бы помочь в отделении вашей программной/бизнес(данных) логики от вашего UI (интерфейса пользователя) кода. Это позволяет более лекго производить модульное тестирование программной/бизнес(данных) логики изолированно от логики генерирования интерфейса пользователя. Это также помогает сделать ваше приложение более удобным в сопровождении - так как делает более проблематичным для вас случайно добавить программную/бизнес(данных) логику в ваш шаблон представления.   
Когда применяем три метода действия нашего ProductController класса, мы будем использовать входящие значения URL параметра для получения соответствующих моделей объектов из нашей базы данных, и затем выбирать компонент "View" (Представление) для возврата необходимого HTML ответа. Мы будем использовать один из RenderView() методов базового класса Контроллера для указания Представления которое мы хотим использовать, а так же явно передавать нужные данные которые мы хотим чтобы Представление использовало в созданном ответе:   
Ниже представлен окончательный вариант реализации нашего класса ProductController:   
Заметьте что количество строк кода в методах верхнего примера достаточно мало (по две строчки каждый). Это отчасти от того что логика URL преобразования параметров полностью реализована за нас платформой MVC (избавляя нас от написания множества кода). От чати это и от того что сценарий отображения достаточно прост с точки зрения бизнес логики (методы действия только выполняют роль чтения-отображения).   
В большинстве случаев однако, вы обнаружите то, что мы называем "тощими контроллерами" ("skinny controllers") что означает методы контроллеров с малым количеством строк кода (менее 10 строк). Чаще всего это хороший знак того что у вас присутствует четкая инкапсуляция логики данных и продуманная логика Контроллера.

**Модульное тестирование (Unit Testing) нашего ProductsController**

Вы возможно удивитесь что следующий шаг который мы собираемся проработать, заключается в том чтобы протестировать нашу программную логику и функциональность. Наверное даже спросите - как такое вообще возможно? Мы не применили наши Представления, и наше приложение в данный момент не производит ни одного HTML тэга. Ну что же, одна из вещей которая делает MVC концепцию такой привлекательной в том, что мы можем проводит модульное тестирование Контроллера, и бизнес логики приложения, абсолютно не зависимо от логики создания View/Htm. Ниже вы увидите что мы можем проводить модульное тестирование даже до того как создадим Представления.   
Для модульного тестирования ProductsController класса, над которым мы работаем, мы добавим ProductsControllerTest класс в Test Project который был добавлен в наше решение по умолчанию когда мы создали ASP.NET MVC Application используя Visual Studio:   
  
Затем мы напишем простой модульный тест который тестирует действие Detail нашего ProductsController:   
ASP.NET MVC каркас был спроектирован специально с возможностью легкого модульного тестирования. Все внутренние интерфейсы процедур (API) и контракты внутри каркаса являются интерфейсами, and extensibility points are provided to enable easy injection and customization of objects (включая возможность использования IOC контейнеров, типа Windsor, StructureMap, Spring.NET, и ObjectBuilder). Разработчики смогут использовать встроенные mock классы, или использовать любой .NET type-mocking framework для создания их собственной тестовой версии родственного MVC объекта.    
В модульном тесте показанном выше , вы можите видеть пример того как мы включаем тестовую "ViewFactory" в наш ProductsController класс перед тем как вызвать метод действия Detail(). Делая так мы переназначаем ViewFactory по умолчанию которая иначе бы управляла созданием и выводом нашего Представления. Мы можем использовать эту тестовую реализацию ViewFactory что бы изолировать тестирование в рамках нашего ProductController's Detail действия (без необходимости запускать для этого Представление). Обратите внимание на то, как затем мы используем три оператора Assert после Detail() метода, вызывая их для того чтобы убедиться в правильности работы (конкретно в том что действие получает правильный объект Product и затем посылает в соответствующее Представление).   
Так как мы можем подделывать (mock) и симулировать любой объект в каркасе MVC (включая объекты IHttpRequest и IHttpResponse) , вам нет необходимости проводить модульное тестирование в контексте реального веб-сервера. Вместо этого мы можим создать наш ProductsController в пределах обычной библиотеки класса и тестировать его напрямую. Это может значительно ускорить выполнение юнит-тестов, так же как упростить их конфигурирование и выполнение.   
Если мы используем Visual Studio 2008 IDE, мы так же можем легко проследить успешность/провал запусков наших тестов (эта возможность теперь встроена в VS 2008 Professional):   
Думаю что вы найдете что ASP.NET MVC Framework делает написание юнит-тестов простым и предоставляет удобный иструмент программирование на основе тестов (TDD workflow).

**Прорисовка интерфейса пользователя (UI) при помощи Отображений**

Мы закончили реализацию и тестирование приложения и логики данных для секции просмотра продукции нашего приложения электронной коммерции. Теперь мы должны реализовать HTML UI для него.   
Мы сделаем это реализовав "Views" которые прорисуют соответствующий UI используя относящиеся к отображению объекты данных которые наш метод выполнения ProductsController передает когда вызывает RenderView() метод:   
На верхнем примере кода параметр "Categories" метода RenderView отображает имя Представления которое мы хотим отобразить, а второй параметр это список объектов категорий, который мы хотим передать в Представление и использовать его как данные для генерации необходимого HTML UI.   
ASP.NET MVC Framework поддерживает возможность использования любого движка для парсинга шаблонов помогающего в создании UI (включая существующие NVelocity, Brail, так же как любой другой написанный вами). По умолчанию ASP.NET MVC Framework использует стандартную ASP.NET Page (.aspx), Master Page (.master), и UserControl (.ascx) поддержку входящую в ASP.NET.  
Мы будем использовать встроенный в ASP.NET движок представления для разработки нашего UI приложения электронной коммерции.

**Создание Site.Master файла**

Так как мы будем делать большое количество страниц для нашего сайта, мы начнем нашу работу над UI с определения базовой страницы которую мы сможем использовать для определения основного HTML шаблона нашего сайта. Мы сделаем это в файле под названием "Site.Master" который мы создадим в папке \Views\Shared нашего проекта:   
Мы можем ссылаться на внешние CSS таблицы стилей, для включения всех стилей сайта и затем использовать базовую страницу для определения общего шаблона сайта, а так же определить регионы размещения информации в которых мы хотели бы дать возможность страницам выводить соответствующие содержимое. При желании мы можем использовать все новые крутые возможности дизайна в VS 2008 (включая [HTML split-view designer](http://weblogs.asp.net/scottgu/archive/2007/07/30/enabling-vertical-split-view-in-vs-2008.aspx), [CSS Authoring](http://weblogs.asp.net/scottgu/archive/2007/07/25/vs-2008-web-designer-and-css-support.aspx), и [Nested Master Pages support](http://weblogs.asp.net/scottgu/archive/2007/07/09/vs-2008-nested-master-page-support.aspx)).

**Понимание структуры папки /Views**

По умолчанию когда вы создаете ASP.NET MVC проект используя Visual Studio, студия создаст подкаталог "Shared" внутри каталога "Views". Это рекомендованное место для хранения Master Pages, User Controls, и Views которые мы хотим открыть для доступа множества Контроллеров нашего приложения.   
Когда мы создаем Представление которое предназначено для конкретного Контроллера, в ASP.NET MVC соглашение по умолчанию хранить их в подкаталоге в корне \Views. По умолчанию имя подкаталога должно соответствовать имени Контроллера. Например, так как класс контроллера который мы создавали назывался "ProductsController", мы разместим все что относиться к Представлению в подкаталог \Views\Products:   
Когда мы вызываем метод Render() внутри конкретного Контроллера, MVC Framework автоматически будет искать соответствующий шаблон .aspx или .ascx внутри папки \Views\ControllerName, и затем если не сможет найти подходящий шаблон Представления там, проверит папку \Views\Shared на наличие последнего.

**Создание Представления Categories**

Мы можем создать Представление "Categories" для нашего ProductsController при помощи Visual Studio используя пункт меню "Add New Item" в каталоге Products и выбрав пункт "MVC View Page" шаблон. Это создаст новую .aspx страницу которую при желании мы можем ассоциировать с нашей Site.Master основной страницей для выбора общего вида и заполнения (и так же как с базовой вы получаете полную WYSIWYG поддержку дизайнера):   
Когда вы создаете приложение используя MVC шаблон, вы должны содержать код Представления максимально простым, и убедиться что код Представления занимается только отрисовкой UI. Логика приложения и получения данных должна быть написана только внутри класса Контроллера. Классы Контроллеров в свою очередь могут выбирать необходимые объекты данных для отрисовки Представления когда они вызывают их RenderView метод. Для нижнего примера в методе действия Categotries нашего ProductsController класса мы передаем список объектов Category в Представление Categories:   
  
Страницы Представления в MVC по умолчанию наследуются от базового класса System.Web.Mvc.ViewPage, который представляет ряд специфичных для MVC вспомогательных методов и свойств которые мы можем использовать в создании нашего UI. Одно из таких свойств называется "ViewData", и оно предоставляет доступ к предназначенным для Представления объектам которые Контроллер передает в качестве аргументов метода RenderView().   
Из своего Представления вы можите получить доступ "ViewData" как в режими позднего связывания так и в строго типизированном. Если ваше Представление наследуется от ViewPage - свойство ViewData будет иметь тип библиотеки позднего связывания. Если ваше Представление наследуется от типизированного базового класса ViewPage где T отображает тип объекта данных ViewData переденного Контроллером в Представление - тогда свойство ViewData будет иметь конкретный тип совпадающий с типом переданным Контроллером.   
Например, показанный ниже класс представления Categories наследуется от ViewPage - там где я выделил Т является объектом List типа Category:   
Это означает что я получаю полную типовую безопасность, [intellisense](http://msdn.microsoft.com/library/rus/default.asp?url=/library/RUS/vsintro7/html/vcovrAutomaticStatementCompletion.asp), и проверку ошибок на этапе компилирования когда работаю с List:

**Вывод нашего Представления Categories**

Если вы помните из самого первого скрин шота этой статьи , мы хотели отобразить список категорий продуктов внутри нашего Представления Categories:   
Я могу написать этот код генерирующий HTML UI одним из двух способов внутри моей реализации Представления. 1) Используя встроенный в .aspx код (Inline Code) , или 2) Используя серверные элементы управления в .aspx файле и привязкой из Code Behind.

**Вывод информации, подход 1: Использование встроенного кода**

ASP.NET Pages, User Controls и Master Pages сегодня поддерживают <% %> и <%= %> синтаксис для встраивания кода в HTML разметку. Мы можем использовать эту технику в нашем Представлении Categories для простого написания цикла foreach который генерирует маркированный HTML список категорий:   
  
VS 2008 поддерживает полный [intellisense](http://msdn.microsoft.com/library/rus/default.asp?url=/library/RUS/vsintro7/html/vcovrAutomaticStatementCompletion.asp) кода внутри редактора для обоих языков VB и C#. Это значит мы получаем intellisense по отношению к нашим объектам Category переданных в представление:   
VS 2008 также предоставляет полную поддержку отладки встроенного кода (позволяет нам устанавливать контрольные точки и динамически изучать любую информацию в Представлении при помощи отладчика):

**Вывод информации, подход 2: Использование серверных элементов**

ASP.NET Pages(страницы), User Controls (элементы пользователя) and Master Pages (базовые страницы) также поддерживают возможность использования встроенных серверных элементов для инкапсуляции генерирования HTML UI. Вместо того чтобы использовать встроенный код, как на примере выше, мы можем использовать новый [элемент входящий в .NET 3.5](http://weblogs.asp.net/scottgu/archive/2007/08/10/the-asp-listview-control-part-1-building-a-product-listing-page-with-clean-css-ui.aspx) для генерирования нашего маркированного HTML списка:   
Обратите внимание на то как элемент ListView инкапсулирует как вывод списка значений так и обработку сценария когда в списке нет не одного элемента ( освобождает нас от необходимости писать в разметке оператор if/else). Затем мы можем привязать наш список объектов категорий к элементу listview используя code-behind код вроде этого:   
*Важно: В мире MVC мы должны располагать логику вывода только code-behind классе нашего Представления (и ни какой логико приложения или данных). Обратите внимание как на верхнем примере мы назначаеем строго типизированному коллекцию ViewData объекта Category элементу ListView. Наш класс контроллера ProductsController тот кто фактически получает список категорий из базы данных, а не Представление.*   
Версия нашего шаблона Представления с применением серверного элемента ListView регенерирует абсолютно такой же HTML код как наша построчная версия. Так как у нас нет   
элемента на странице и нет viewstate, значения ID или другой разметки будут пустые. Только чистый CSS friendly HTML:

**Метод Html.ActionLink**

Одна из вещей на которую вы наверное обратили внимание в обоих построчной и сервер-компонентной версиях примеров нашего Представления представленных выше, это вызовы метода Html.ActionLink: Html объект является вспомогательным свойством базового класса ViewPage, а метод ActionLink его помошник, который делает простым динамическое создание HTML гиперссылки которая указывает обратно на метод действия Контроллера. Если вы посмотрите на HTML вывод верхнего скрин шота, вы можите увидеть несколько примеров HTML кода сгенерированного этим методом:<а hrеf="http://weblogs.asp.net/Products/List/Beverages">BeveragesТак выглядит запись вспомогательного метода Html.ActionLink которую я использую:   
string ActionLink(string text, object values);   
Первый аргумент представляет собой содержание гиперссылки для вывода (например: идем сюда). Второй аргумент является [анонимным объектом](http://weblogs.asp.net/scottgu/archive/2007/05/15/new-orcas-language-feature-anonymous-types.aspx?CommentPosted=true) который представляет последовательность значений используемых для создания конкретного URL (можете считать это [ловким способом создания каталогов](http://weblogs.asp.net/leftslipper/archive/2007/09/24/using-c-3-0-anonymous-types-as-dictionaries.aspx)). Я рассмотрю более детально как это все работает в следующей статье охватывающей работу движка распределения URL. Основная мысль в том, что вы можите использовать систему URL распределения как для обработки входящих URL так и для создания URL для вывода с HTML кодом. Если у нас есть правило распределения вроде этого:  
И мы напишем такой код в Представлении Category:   
<%= Html.ActionLink("Click Me to See Beverages", new { action="List", category="Beverages" } %>   
Метод ActionLink использует правила URL распределения вашего приложения для загрузки ваших параметров и генерирует такой вывод:   
<а hrеf="http://weblogs.asp.net/Products/List/Beverages">Click Me to See Beverages   
Это дает возможность упростить в вашем приложении обратные вызовы URLs и AJAX к Контроллеру. Это так же значит что вы можите изменить правила URL распределения в одном месте и код во всем приложении автоматически примет изменения как для обработки входящих URL так и генерирования исходящих URL.   
*Важное замечание: В целях улучшения тестирования, MVC framework сегодня не поддерживает отправку сообщений напрямую серверным компонентам ванего Представления. Вместо этого ASP.NET MVC приложение генерирует гиперссылку и AJAX вызов методу Контроллера - и замет Представление (и с ним любой серверный элемент) исключительно для отображения вывода. Это помогает убедиться в том что логика вашего Представления минимальна и полностью сфокусирована на отображении и что вы можите легко проводить модульное тестирование вашего класса Контроллера, и быть уверенным что вся логика приложения и данных независима от Представления. В следующий раз я более подробно напишу об этом.*

**Вывод**

Эта первая статья получилась достаточно длинной, однако надеюсь дала достаточно широкий обзор того, как разные компоненты нового ASP.NET MVC Framework работают вместе, и как вы можите выполнять при помощи их разнообразные задачи. Первый публичный обзор будет доступен в течении нескольких недель и вы сможете использовать его для выполнения всех вышеописанных действий.   
Многие концепции присущие MVC (особенно идея разделения ролей) возможно новы для многих людей читающих эту статью, надеюсь что эта статья показала что реализация ASP.NET MVC над которой мы работали достаточно хорошо вписывается в имеющиеся возможности ASP.NET, .NET, и Visual Studio. Вы можете использовать .ASPX, .ASCX и .MASTER файлы и ASP.NET AJAX для создания Отображений ASP.NET MVC. Возможности не относящиеся к UI в ASP.NET сегодня, такие как Forms Authentication, Windows Authentication, Membership, Roles, Url Authorization, Caching, Session State, Profiles, Health Monitoring, Configuration, Compilation, Localization, and HttpModules/HttpHandlers полностью поддерживают модель MVC.   
Если вам не нравится модель MVC, или вы находите ее не естественным для вашего стиля программирования, вы конечно же можите ее не использовать. Это полностью дополнительная возможность которая и ни коим образом не вытесняет существующую модель WebForms Page Controller. Как WebForms так и MVC будут полностью поддерживаться и усовершенствоваться по мере развития. При желании вы даже можете создать одно приложение часть которого будет написана используя WebForms а другая использовать подход MVC.   
Если вам по душе то что вы увидели в этой статье об MVC (или заинтригованы и хотите узнать больше) , следите за моим блогом на бушующей неделе. Я буду рассказывать больше о возможностях MVC и использовать их для построения приложения электронной коммерции.   
Надеюсь это помогло,

**Пример асинхронного сокета сервера**

* 30.03.2017
* Время чтения: 6 мин
* Соавторы
  + [Karel Zikmund](https://github.com/karelz)
  + [](https://github.com/olprod)
  + [](https://github.com/OpenLocalizationService)
  + [Saisang Cai](https://github.com/Saisang)

Приведенная ниже программа создает сервер, который принимает запросы на соединение от клиентов. Сервер создается с использованием асинхронного сокета, поэтому выполнение серверного приложения не приостанавливается, пока оно ожидает подключения от клиента. Приложение получает строку от клиента, выводит ее в консоли, а затем отправляет строку обратно клиенту. В строке, полученной от клиента, конец сообщения должен быть помечен строкой "<EOF>".

C#

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

// State object for reading client data asynchronously

public class StateObject {

// Client socket.

public Socket workSocket = null;

// Size of receive buffer.

public const int BufferSize = 1024;

// Receive buffer.

public byte[] buffer = new byte[BufferSize];

// Received data string.

public StringBuilder sb = new StringBuilder();

}

public class AsynchronousSocketListener {

// Thread signal.

public static ManualResetEvent allDone = new ManualResetEvent(false);

public AsynchronousSocketListener() {

}

public static void StartListening() {

// Establish the local endpoint for the socket.

// The DNS name of the computer

// running the listener is "host.contoso.com".

IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());

IPAddress ipAddress = ipHostInfo.AddressList[0];

IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

// Create a TCP/IP socket.

Socket listener = new Socket(ipAddress.AddressFamily,

SocketType.Stream, ProtocolType.Tcp );

// Bind the socket to the local endpoint and listen for incoming connections.

try {

listener.Bind(localEndPoint);

listener.Listen(100);

while (true) {

// Set the event to nonsignaled state.

allDone.Reset();

// Start an asynchronous socket to listen for connections.

Console.WriteLine("Waiting for a connection...");

listener.BeginAccept(

new AsyncCallback(AcceptCallback),

listener );

// Wait until a connection is made before continuing.

allDone.WaitOne();

}

} catch (Exception e) {

Console.WriteLine(e.ToString());

}

Console.WriteLine("\nPress ENTER to continue...");

Console.Read();

}

public static void AcceptCallback(IAsyncResult ar) {

// Signal the main thread to continue.

allDone.Set();

// Get the socket that handles the client request.

Socket listener = (Socket) ar.AsyncState;

Socket handler = listener.EndAccept(ar);

// Create the state object.

StateObject state = new StateObject();

state.workSocket = handler;

handler.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,

new AsyncCallback(ReadCallback), state);

}

public static void ReadCallback(IAsyncResult ar) {

String content = String.Empty;

// Retrieve the state object and the handler socket

// from the asynchronous state object.

StateObject state = (StateObject) ar.AsyncState;

Socket handler = state.workSocket;

// Read data from the client socket.

int bytesRead = handler.EndReceive(ar);

if (bytesRead > 0) {

// There might be more data, so store the data received so far.

state.sb.Append(Encoding.ASCII.GetString(

state.buffer, 0, bytesRead));

// Check for end-of-file tag. If it is not there, read

// more data.

content = state.sb.ToString();

if (content.IndexOf("<EOF>") > -1) {

// All the data has been read from the

// client. Display it on the console.

Console.WriteLine("Read {0} bytes from socket. \n Data : {1}",

content.Length, content );

// Echo the data back to the client.

Send(handler, content);

} else {

// Not all data received. Get more.

handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,

new AsyncCallback(ReadCallback), state);

}

}

}

private static void Send(Socket handler, String data) {

// Convert the string data to byte data using ASCII encoding.

byte[] byteData = Encoding.ASCII.GetBytes(data);

// Begin sending the data to the remote device.

handler.BeginSend(byteData, 0, byteData.Length, 0,

new AsyncCallback(SendCallback), handler);

}

private static void SendCallback(IAsyncResult ar) {

try {

// Retrieve the socket from the state object.

Socket handler = (Socket) ar.AsyncState;

// Complete sending the data to the remote device.

int bytesSent = handler.EndSend(ar);

Console.WriteLine("Sent {0} bytes to client.", bytesSent);

handler.Shutdown(SocketShutdown.Both);

handler.Close();

} catch (Exception e) {

Console.WriteLine(e.ToString());

}

}

public static int Main(String[] args) {

StartListening();

return 0;

}

}

## См. также

[Примеры асинхронных сокетов клиента](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/asynchronous-client-socket-example)  
[Использование асинхронных сокетов сервера](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/using-an-asynchronous-server-socket)  
[Примеры кода сокетов](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/socket-code-examples)

**Пример синхронного сокета клиента**

Приведенная ниже программа создает клиент, который подключается к серверу. Клиент создается с использованием синхронного сокета, поэтому выполнение клиентского приложения приостанавливается до тех пор, пока сервер не вернет ответ. Приложение отправляет строку на сервер, а затем выводит возвращенную им строку в консоли.

C#

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

public class SynchronousSocketClient {

public static void StartClient() {

// Data buffer for incoming data.

byte[] bytes = new byte[1024];

// Connect to a remote device.

try {

// Establish the remote endpoint for the socket.

// This example uses port 11000 on the local computer.

IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());

IPAddress ipAddress = ipHostInfo.AddressList[0];

IPEndPoint remoteEP = new IPEndPoint(ipAddress,11000);

// Create a TCP/IP socket.

Socket sender = new Socket(ipAddress.AddressFamily,

SocketType.Stream, ProtocolType.Tcp );

// Connect the socket to the remote endpoint. Catch any errors.

try {

sender.Connect(remoteEP);

Console.WriteLine("Socket connected to {0}",

sender.RemoteEndPoint.ToString());

// Encode the data string into a byte array.

byte[] msg = Encoding.ASCII.GetBytes("This is a test<EOF>");

// Send the data through the socket.

int bytesSent = sender.Send(msg);

// Receive the response from the remote device.

int bytesRec = sender.Receive(bytes);

Console.WriteLine("Echoed test = {0}",

Encoding.ASCII.GetString(bytes,0,bytesRec));

// Release the socket.

sender.Shutdown(SocketShutdown.Both);

sender.Close();

} catch (ArgumentNullException ane) {

Console.WriteLine("ArgumentNullException : {0}",ane.ToString());

} catch (SocketException se) {

Console.WriteLine("SocketException : {0}",se.ToString());

} catch (Exception e) {

Console.WriteLine("Unexpected exception : {0}", e.ToString());

}

} catch (Exception e) {

Console.WriteLine( e.ToString());

}

}

public static int Main(String[] args) {

StartClient();

return 0;

}

}

**См. также**

[Пример синхронного сокета сервера](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/synchronous-server-socket-example)  
[Использование синхронного сокета клиента](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/using-a-synchronous-client-socket)  
[Примеры кода сокетов](https://docs.microsoft.com/ru-ru/dotnet/framework/network-programming/socket-code-examples)

**Сокеты**

[C# и .NET](https://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php) --- [Сетевое программирование](https://professorweb.ru/my/csharp/web/level1/web_index.php) --- Сокеты

***Сокет*** — это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами (локальными или удаленными). Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Первоначально сокеты были разработаны для UNIX в Калифорнийском университете в Беркли. В UNIX обеспечивающий связь метод ввода-вывода следует алгоритму open/read/write/close. Прежде чем ресурс использовать, его нужно открыть, задав соответствующие разрешения и другие параметры. Как только ресурс открыт, из него можно считывать или в него записывать данные. После использования ресурса пользователь должен вызывать метод Close(), чтобы подать сигнал операционной системе о завершении его работы с этим ресурсом.

Когда в операционную систему UNIX были добавлены средства *межпроцессного взаимодействия (Inter-Process Communication, IPC)* и сетевого обмена, был заимствован привычный шаблон ввода-вывода. Все ресурсы, открытые для связи, в UNIX и Windows идентифицируются дескрипторами. Эти дескрипторы, или *описатели (handles)*, могут указывать на файл, память или какой-либо другой канал связи, а фактически указывают на внутреннюю структуру данных, используемую операционной системой. Сокет, будучи таким же ресурсом, тоже представляется дескриптором. Следовательно, для сокетов жизнь дескриптора можно разделить на три фазы: открыть (создать) сокет, получить из сокета или отправить сокету и в конце концов закрыть сокет.

Интерфейс IPC для взаимодействия между разными процессами построен поверх методов ввода-вывода. Они облегчают для сокетов отправку и получение данных. Каждый целевой объект задается адресом сокета, следовательно, этот адрес можно указать в клиенте, чтобы установить соединение с целью.

**Типы сокетов**

Существуют два основных типа сокетов — потоковые сокеты и дейтаграммные.

**Потоковые сокеты (stream socket)**

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т, е. через эту конечную точку приложение может и передавать, и получать данные.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных. Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потоковые сокеты достигают этого уровня качества за счет использования протокола ***Transmission Control Protocol (TCP)***. TCP обеспечивает поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают. Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако, отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании TCP этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: сокет А запрашивает соединение с сокетом В, а сокет В либо соглашается с запросом на установление соединения, либо отвергает его.

Если данные должны гарантированно доставляться другой стороне или размер их велик, потоковые сокеты предпочтительнее дейтаграммных. Следовательно, если надежность связи между двумя приложениями имеет первостепенное значение, выбирайте потоковые сокеты.

Сервер электронной почты представляет пример приложения, которое должно доставлять содержание в правильном порядке, без дублирования и пропусков. Потоковый сокет рассчитывает, что TCP обеспечит доставку сообщений по их назначениям.

**Дейтаграммные сокеты (datagram socket)**

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы (например, сервер времени суток, обеспечивающий синхронизацию времени для своих клиентов). В конце концов на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание, и задача серверного приложения не выполняется. Для сокращения накладных расходов нужно использовать дейтаграммные сокеты.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался ***User Datagram Protocol (UDP)***. В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

Кроме двух рассмотренных типов существует также обобщенная форма сокетов, которую называют необрабатываемыми или сырыми.

**Сырые сокеты (raw socket)**

Главная цель использования сырых сокетов состоит в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Это достигается обеспечением специальной реализации стека TCP/IP, замещающей механизм, предоставленный стеком TCP/IP в ядре — пакет непосредственно передается приложению и, следовательно, обрабатывается гораздо эффективнее, чем при проходе через главный стек протоколов клиента.

По определению, *сырой сокет* — это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению.

При использовании таких сокетов пакет не проходит через фильтр TCP/IP, т.е. никак не обрабатывается, и предстает в своей сырой форме. В таком случае обязанность правильно обработать все данные и выполнить такие действия, как удаление заголовков и разбор полей, ложится на получающее приложение — все равно, что включить в приложение небольшой стек TCP/IP.

Однако нечасто может потребоваться программа, работающая с сырыми сокетами. Если вы не пишете системное программное обеспечение или программу, аналогичную анализатору пакетов, вникать в такие детали не придется. Сырые сокеты главным образом используются при разработке специализированных низкоуровневых протокольных приложений. Например, такие разнообразные утилиты TCP/IP, как trace route, ping или arp, используют сырые сокеты.

Работа с сырыми сокетами требует солидного знания базовых протоколов TCP/UDP/IP.

**Порты**

Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным номером порта, определенным при установлении соединения.

Сокет состоит из IP-адреса машины и номера порта, используемого приложением TCP. Поскольку IP-адрес уникален в Интернете, а номера портов уникальны на отдельной машине, номера сокетов также уникальны во всем Интернете. Эта характеристика позволяет процессу общаться через сеть с другим процессом исключительно на основании номера сокета.

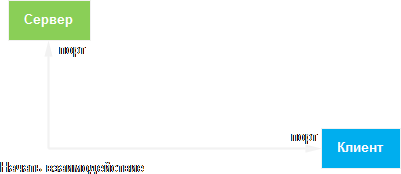
За определенными службами номера портов зарезервированы — это широко известные номера портов, например порт 21, использующийся в FTP. Ваше приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят. Агентство *Internet Assigned Numbers Authority (IANA)* ведет перечень широко известных номеров портов.

Обычно приложение клиент-сервер, использующее сокеты, состоит из двух разных приложений - клиента, инициирующего соединение с целью (сервером), и сервера, ожидающего соединения от клиента.

Например, на стороне клиента, приложение должно знать адрес цели и номер порта. Отправляя запрос на соединение, клиент пытается установить соединение с сервером:

Запрос клиента серверу

Если события развиваются удачно, при условии что сервер запущен прежде, чем клиент попытался с ним соединиться, сервер соглашается на соединение. Дав согласие, серверное приложение создает новый сокет для взаимодействия именно с установившим соединение клиентом:



Теперь клиент и сервер могут взаимодействовать между собой, считывая сообщения каждый из своего сокета и, соответственно, записывая сообщения.

**Работа с сокетами в .NET**

Поддержку сокетов в .NET обеспечивают классы в пространстве имен **System.Net.Sockets** — начнем с их краткого описания.

|  |  |
| --- | --- |
| Классы для работы с сокетами | |
| **Класс** | **Описание** |
| *MulticastOption* | Класс MulticastOption устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее. |
| *NetworkStream* | Класс NetworkStream реализует базовый класс потока, из которого данные отправляются и в котором они получаются. Это абстракция высокого уровня, представляющая соединение с каналом связи TCP/IP. |
| *TcpClient* | Класс TcpClient строится на классе Socket, чтобы обеспечить TCP-обслуживание на более высоком уровне. TcpClient предоставляет несколько методов для отправки и получения данных через сеть. |
| *TcpListener* | Этот класс также построен на низкоуровневом классе Socket. Его основное назначение — серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях. |
| *UdpClient* | UDP — это протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET требуется другая функциональность. |
| *SocketException* | Это исключение порождается, когда в сокете возникает ошибка. |
| *Socket* | Последний класс в пространстве имен System.Net.Sockets — это сам класс Socket. Он обеспечивает базовую функциональность приложения сокета. |

**Класс Socket**

Класс Socket играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Главным образом, вызовы методов этого класса выполняют необходимые проверки, связанные с безопасностью, в том числе проверяют разрешения системы безопасности, после чего они переправляются к аналогам этих методов в Windows Sockets API.

Прежде чем обращаться к примеру использования класса Socket, рассмотрим некоторые важные свойства и методы этого класса:

|  |  |
| --- | --- |
| Свойства и методы класса Socket | |
| **Свойство или метод** | **Описание** |
| AddressFamily | Дает семейство адресов сокета — значение из перечисления Socket.AddressFamily. |
| Available | Возвращает объем доступных для чтения данных. |
| Blocking | Дает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме. |
| Connected | Возвращает значение, информирующее, соединен ли сокет с удаленным хостом. |
| LocalEndPoint | Дает локальную конечную точку. |
| ProtocolType | Дает тип протокола сокета. |
| RemoteEndPoint | Дает удаленную конечную точку сокета. |
| SocketType | Дает тип сокета. |
| Accept() | Создает новый сокет для обработки входящего запроса на соединение. |
| Bind() | Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение. |
| Close() | Заставляет сокет закрыться. |
| Connect() | Устанавливает соединение с удаленным хостом. |
| GetSocketOption() | Возвращает значение SocketOption. |
| IOControl() | Устанавливает для сокета низкоуровневые режимы работы. Этот метод обеспечивает низкоуровневый доступ к лежащему в основе классу Socket. |
| Listen() | Помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений. |
| Receive() | Получает данные от соединенного сокета. |
| Poll() | Определяет статус сокета. |
| Select() | Проверяет статус одного или нескольких сокетов. |
| Send() | Отправляет данные соединенному сокету. |
| SetSocketOption() | Устанавливает опцию сокета. |
| Shutdown() | Запрещает операции отправки и получения данных на сокете. |