7) Javascript, понятие о AJAX и JSONP // Основные концепции ASP.NET MVC

Javascript, понятие о AJAX и JSONP

JavaScript — прототипно-ориентированный сценарный язык программирования.

Вставка (внедрение, подключение) кода JavaScript в страницу (разметку) ASP.NET

JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Основные архитектурные черты: динамическая типизация, слабая типизация, автоматическое управление памятью, прототипное программирование, функции как объекты первого класса.

Для внедрения клиентского кода JavaScript в разметку страницы, генерируемой c# существует множество подходов.

Первый способ - самый любимый новичками в ASP.NET

В бихайн-коде в требуемом месте записываем инструкцию Response.Write

```
string java = “Some JavaScript code”;

Response.Write(java);
```

Но, во первых не возможно управлять местом вывода параметра, что иногда чрезвычайно важно, а во вторых – плывут стили на странице. В итоге, вся разметка «съезжает и располэается по швам».

Другой вариант более управляемый

В разметке страницы, к примеру, в теге body вставляем блок серверного кода

```
<body>
...
<%=java %>
...
</body>
```

а в бихайн коде где-нибудь, где это необходимо, устанавливаем значение в переменную:

```
protected string java;
void Page_Load(object sender, EventArgs e)
{
    java = “Some JavaScript code”;
}
```

Этот подход прекрасно работает, но все ж он не соответствует природе ООР и больше напоминает о технологии ASP.

Более правильным будет третий подход

В данном случае, целесообразно воспользоваться встроенными функциями платформы .NET, а именно методом регистрации клиентских скриптов Page.ClientScript.RegisterClientScriptBlock:

```
Page.ClientScript.RegisterClientScriptBlock(this.GetType(), “clientScript”, java);
```

На последок хочу показать, как при нажатии на кнопке предварительно получать клиентское окно с подтверждением желаемых действий. К примеру, на странице имеется кнопка, по нажатию на которой серверная сторона производит удаление запи из БД. Естественно, перед удалением желательно переспросить клиента о желании удалять запись. Но перезагружать страницу с целью вывода запроса на подтверждение действия – очень плохая идея. Как я уже сказал, лучше на клиенте переспросить и уже исходя из полученных результатов отправлять данные на сервер, или нет.

Итак, у любой кнопки есть два свойства **OnClick** и **OnClientClick**

```
<body>
<asp:Button ID="Mybtn" runat="server" onclick="Mybtn_Click"
onclientclick="return ConfirmAction()" Text="Удалить запись" />

<script type="text/javascript" language="javascript">
function ConfirmAction ()
{
    return confirm('Подтверждаете удаление?');
}
</script>
</body>
```

При нажатии на кнопку появляется окно подтверждения действий. При нажатии на кнопку "OK" происходит отправка страницы на сервер и вызов метода Mybtn_Click. При отмене, ничего не происходит – окно сворачивается и страница продолжает пребывать в браузере.

В данном простом случае вообще нет смысла регистрировать клиентскую функцию, можно применить следующую запись:

```
<asp:Button ID="Button1" runat="server" onclick="Mybtn_Click"
onclientclick="javascript:
   return confirm('Сохранить настройки?');"
Text="Удалить запись" />
```

Клиентский JavaScript-код может встраиваться в HTML-документы четырьмя способами:

- встроенные сценарии между парой тегов <script> и </script>;
- из внешнего файла, заданного атрибутом src тега <script>;
- в обработчик события, заданный в качестве значения HTML-атрибута, такого как onclick или onmouseover;
- как тело URL-адреса, использующего специальный спецификатор псевдопротокола JavaScript:.

В следующих далее подразделах описываются все четыре способа встраивания программного кода на языке JavaScript. Следует отметить, что HTML-атрибуты обработчиков событий и адреса URL с псевдопротоколом javascript: редко используются в современной практике программирования на языке JavaScript (они были более распространены на раннем этапе развития Всемирной паутины).

Встроенные сценарии (в тегах <script> без атрибута src) также стали реже использоваться по сравнению с прошлым. Согласно философии программирования, известной как *ненавязчивый JavaScript (unobtrusive JavaScript)*, содержимое (разметка HTML) и поведение (программный код на языке JavaScript) должны быть максимально отделены друг от друга. Следуя этой философии программирования, сценарии на языке JavaScript лучше встраивать в HTML-документы с помощью элементов <script>, имеющих атрибут src.

Элемент <script>

Клиентские JavaScript-сценарии могут встраиваться в HTML-файлы между тегами <script> и </script>:

```
<script>
        // Здесь располагается JavaScript-код
</script>
```

В языке разметки XHTML содержимое тега <script> обрабатывается наравне с содержимым любого другого тега. Если JavaScript-код содержит символы < или &, они интерпретируются как элементы XML-разметки. Поэтому в случае применения языка XHTML лучше помещать весь JavaScript-код внутрь секции CDATA:

```
<script><![CDATA[
        // Здесь располагается JavaScript-код
]]></script>
```

Сценарии во внешних файлах

Тег <script> поддерживает атрибут src, который определяет URL-адрес файла, содержащего JavaScript-код. Используется он следующим образом:

```
<script src="../../scripts/myscript.js"></script>
```

Файл JavaScript-кода обычно имеет расширение .js и содержит JavaScript-код в «чистом виде» без тегов <script> или любого другого HTML-кода.

Использование тега с атрибутом src дает ряд преимуществ:

- HTML-файлы становятся проще, т.к. из них можно убрать большие блоки JavaScript-кода, что помогает отделить содержимое от поведения.
- JavaScript-функцию или другой JavaScript-код, используемый несколькими HTML-файлами, можно держать в одном файле и считывать при необходимости. Это уменьшает объем занимаемой дисковой памяти и намного облегчает поддержку программного кода, т.к. отпадает необходимость править каждый HTML-файл при изменении кода.
- Если сценарий на языке JavaScript используется сразу несколькими страницами, он будет загружаться браузером только один раз, при первом его использовании - последующие страницы будут извлекать его из кэша браузера.
- Атрибут src принимает в качестве значения произвольный URL-адрес, поэтому JavaScript-программа или веб-страница с одного веб-сервера может воспользоваться кодом (например, из библиотеки подпрограмм), предоставляемым другими веб-серверами. Многие рекламодатели в Интернете используют этот факт.

Обработчики событий в HTML

JavaScript-код, расположенный в теге <script>, исполняется один раз, когда содержащий его HTML-файл считывается в веб-браузер. Для обеспечения интерактивности программы на языке JavaScript должны определять обработчики событий - JavaScript-функции, которые регистрируются в веб-браузере и автоматически вызываются веб-браузером в ответ на определенные события (такие как ввод данных пользователем).

JavaScript-код может регистрировать обработчики событий, присваивая функции свойствам объектов Element (таким как onclick или onmouseover), представляющих HTML-элементы в документе.

Свойства обработчиков событий, такие как onclick, отражают HTML-атрибуты с теми же именами, что позволяет определять обработчики событий, помещая JavaScript-код в HTML-атрибуты. Например:

```
<button onClick="alert('Привет')">Щелкните меня!</button>
```

Обратите внимание на атрибут onClick. JavaScript-код, являющийся значением этого атрибута, будет выполняться всякий раз, когда пользователь будет щелкать на кнопке.

Атрибуты обработчиков событий, включенных в разметку HTML, могут содержать одну или несколько JavaScript-инструкций, отделяемых друг от друга точками с запятой. Эти инструкции будут преобразованы интерпретатором в тело функции, которая в свою очередь станет значением соответствующего свойства обработчика события.

JavaScript в URL

Еще один способ выполнения JavaScript-кода на стороне клиента - включение этого кода в URL-адресе вслед за спецификатором псевдопротокола **javascript:**. Этот

специальный тип протокола обозначает, что тело URL-адреса представляет собою произвольный JavaScript-код, который должен быть выполнен интерпретатором JavaScript. Он интерпретируется как единственная строка, и потому инструкции в ней должны быть отделены друг от друга точками с запятой, а для комментариев следует использовать комбинации символов /* */, а не //.

URL вида javascript: можно использовать везде, где допускается указывать обычные URL: в атрибуте href тега <a>, в атрибуте action тега <form> и даже как аргумент метода, такого как window.open(). Например, адрес URL с программным кодом на языке JavaScript в гиперссылке может иметь такой вид:

```
<a href="JavaScript:new Date().toLocaleTimeString();">
    Который сейчас час?
</a>
```

Когда загружается такой URL-адрес, браузер выполняет JavaScript-код, но, т.к. он не имеет возвращаемого значения (метод alert() возвращает значение undefined), такие браузеры, как Firefox, не затирают текущий отображаемый документ. (В данном случае URL-адрес javascript: служит той же цели, что и обработчик события onclick. Ссылку выше лучше было бы выразить как обработчик события onclick элемента <button> - элемент <a> в целом должен использоваться только для гиперссылок, которые загружают новые документы.)

Если необходимо гарантировать, что URL-адрес javascript: не затрет документ, можно с помощью оператора void обеспечить принудительный возврат значения undefined:

```
<a href="javascript:void window.open('about:blank');">Открыть окно</a>
```

Сокращение AJAX происходит от *Asynchronous JavaScript and XML* (*асинхронный код JavaScript и XML*). Этим общим термином обозначаются высокоинтерактивные веб-приложения, быстро реагирующие на действия пользователя. Основой удаленного исполнения сценариев является возможность отправки асинхронных HTTP-запросов. В данном контексте под *асинхронным вызовом* понимается запрос HTTP, который выдается за пределами встроенного модуля, обеспечивающего отправку форм HTTP. Асинхронный вызов инициируется неким событием HTML-страницы и обслуживается компонентом-посредником. В новейших AJAX-решениях таким посредником является объект `XMLHttpRequest`. Компонент-посредник отправляет обычный запрос HTTP и дожидается - синхронно или асинхронно - завершения его обработки. Получив готовые данные ответа, посредник вызывает JavaScript-функцию обратного вызова. Эта функция должна обновить все части страницы, нуждающиеся в обновлении.

Исторически первая реализация `XMLHttpRequest` появилась в Internet Explorer 5 в виде объекта ActiveX. Затем `XMLHttpRequest` был реализован в других браузерах, но так как ActiveX поддерживается только в Internet Explorer, другие браузеры обычно реализуют `XMLHttpRequest` в виде внутреннего объекта. Эти факты накладывают свой отпечаток на

клиентской процедуре создания объекта `XMLHttpRequest`. Одна из реализаций этой процедуры выглядит следующим образом (код на языке JavaScript):

```
function getXMLHTTP() {
    var XMLHTTP = null;
    if (window.ActiveXObject) {
       try {
           XMLHTTP = new ActiveXObject("Msxml2.XMLHTTP");
       }
       catch (e) {
           try {
               XMLHTTP = new ActiveXObject("Microsoft.XMLHTTP");
           }
           catch (e) { }
       }
    }
    else if (window.XMLHttpRequest) {
            try {
                XMLHTTP = new XMLHttpRequest();
            }
            catch (e) { }
         }
    return XMLHTTP;
}
```

Стандартный набор свойств объекта `XMLHttpRequest`

Свойство	Описание
`readyState`	Числовой код - текущее состояние запроса. Возможные значения: целые числа от 0 до 4. 0 - «Запрос не инициализирован», 1 - «Метод `open()` вызван успешно», 2 - «Метод `send()` вызван успешно», 3 - «Прием данных», 4 - «Ответ получен»
`onreadystatechange`	Функция обратного вызова (событие), вызываемая при изменении `readyState`
`status`	HTTP-статус ответа (например, 200 - *OK*, 400 - *Ресурс не найден*)

statusText	Текстовое описание HTTP-статуса
responseText	Ответ сервера в виде текста
responseXML	Ответ сервера в виде XML-объекта

Из методов объекта XMLHttpRequest наиболее часто применяются два. Метод open() служит для подготовки запроса. Этот метод может принимать до пяти параметров, но обычно используются первых два: тип запроса (обычно "GET" или "POST") и целевой URI. Третий параметр по умолчанию установлен в true, что означает асинхронное поведение (если запрос нужно выполнить синхронно, установите третий параметр в false). Метод send() посылает подготовленный запрос на сервер.

Работая с XMLHttpRequest, программист обычно выполняет следующие четыре задачи:

1. Создает объект XMLHttpRequest, как показано выше.
2. Устанавливает ссылку на функцию в свойстве onreadystatechange.
3. Вызывает у объекта XMLHttpRequest метод open() (подготовка запроса).
4. Посылает запрос, используя метод send().

```
var XMLHTTP = getXMLHTTP();
if (XMLHTTP != null) {
    XMLHTTP.open("GET", "ajax.aspx?sendData=ok");
    XMLHTTP.onreadystatechange = stateChanged;
    XMLHTTP.send(null);
}
function stateChanged()
{
    if (XMLHTTP.readyState == 4 && XMLHTTP.status == 200) {
        window.alert(XMLHTTP.responseText);
    }
}
```

Естественно, что серверный код должен ожидать асинхронный запрос и правильно обработать его. Следующий код демонстрирует метод Page_Load(), который можно использовать как серверный обработчик для предыдущего примера.

```
protected void Page_Load()
{
```

```
    if (Request.QueryString["sendData"] != null &&
        Request.QueryString["sendData"] == "ok")
    {
        Response.Write("Hello from the server!");
        Response.End();
    }
}
```

Предыдущий пример клиентского кода оправлял на сервер GET-запрос. Если планируется отправлять информацию в POST-запросе, нужно установить правильный параметр метода `open()`. Сами данные передаются в виде набора пар как параметр метода `send()`.

```
XMLHTTP.open("POST", "ajax.aspx");
XMLHTTP.onreadystatechange = stateChanged;
XMLHTTP.send("sendData=ok&returnValue=123");
```

В некоторых ситуациях (например, посылка SOAP-запроса веб-службам) методу `send()` передается XML-строка. Однако при этом обычно необходимо указать тип контента:

```
XMLHTTP.open("POST", "ajax.aspx");
XMLHTTP.onreadystatechange = stateChanged;
XMLHTTP.setRequestHeader("Content-Type", "text/xml");
XMLHTTP.send("<soap:Envelope>...</soap:Envelope>");
```

Свойство `responseXML` представляет ответ сервера в виде объекта `XMLDocument`. Предположим, что метод `Page_Load()` из предыдущего примера модифицирован следующим образом (обратите внимание на установку свойства `Response.ContentType`):

```
protected void Page_Load()
{
    if (Request.QueryString["sendData"] != null &&
        Request.QueryString["sendData"] == "ok")
    {
        string xml = "<book title='Programming ASP.NET AJAX'>
                      <chapters>
                        <chapter number='1' title='Introduction' />
```

```
                        <chapter number='2' title='JavaScript' />
                    </chapters></book>";
            Response.ContentType = "text/xml";
            Response.Write(xml);
            Response.End();
        }
    }
```

Следующий JavaScript-код выполняет парсинг XML-информации и формирует содержимое HTML-элемента с тегом `"output"`[1]:

```
var xml = XMLHTTP.responseXML;
var root = xml.documentElement;
document.getElementById("output").innerHTML =
                                        root.getAttribute("title");

var list = document.getElementById("list");
var chapters = xml.getElementsByTagName("chapter");
for (var i=0; i<chapters.length; i++)
{
    var listItem = document.createElement("li");
    var listItemText = document.createTextNode(
                    chapters[i].getAttribute("number") + ": " +
                    chapters[i].getAttribute("title"));
    listItem.appendChild(listItemText);
    list.appendChild(listItem);
}
```

Несмотря на популярность формат XML, многие программисты при работе с AJAX используют формат JSON. *JavaScript Object Notation* – текстовый формат обмена данными. Достоинствами JSON являются простота, компактность и легкая интеграция с JavaScript.

JSON строится на двух структурах:

---

[1] В Internet Explorer манипуляции со структурой документа можно производить только после полной загрузки документа. В этом случае можно код отправки асинхронного события записать в обработчик `window.onload`.

- *Набор пар «имя и значение»*. В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив.
- *Пронумерованный набор значений.* Во многих языках это реализовано как массив, вектор, список или последовательность.

Указанные универсальные структуры данных теоретически поддерживаются (в той или иной форме) всеми современными языками программирования. В JSON используются следующие форматы структур. *Объект* - неупорядоченное множество пар «имя и значение», заключенное в фигурные скобки. Имя отделяется от значения символом ":", а пары разделяются запятыми. *Массив* (одномерный) - множество значений, имеющих порядковые номера (индексы). Массив заключается в квадратные скобки, значения отделяются запятыми. И в объекте, и в массиве значение может быть строкой в двойных кавычках, числом, `true` или `false`, `null`, объектом, массивом. Структуры *объект* и *массив* могут быть вложены друг в друга.

Ниже приведён пример информации, записанной в формате JSON:

```
{"book": {
    "title": "Programming ASP.NET AJAX",
    "author": "Christian Wenz",
    "chapters": [ {"number": "1", "title": "Introduction"},
                  {"number": "2", "title": "JavaScript"} ]
    }
}
```

JSONP или «JSON with padding» (JSON с набивкой) это дополнение к базовому формату JSON. Он предоставляет способ запросить данные с сервера, находящегося в другом домене — операцию, запрещённую в типичных веб-браузерах из-за политики ограничения домена.

По правилу ограничения домена веб-страница, доставляемая с сервера server1.example.com, не может нормально связаться с сервером, отличным от server1.example.com. Исключением является HTML-элемент `<script>` находящийся на данной странице. Эксплуатируя открытую политику для элементов `<script>`, некоторые страницы используют их, чтобы загружать JavaScript-код, оперирующий динамически создаваемыми JSON-данными из других источников. Этот шаблон поведения известен как JSONP. Запросы для JSONP получают не JSON, а произвольный JavaScript-код. Они обрабатываются интерпретатором JavaScript, а не парсером JSON.

Чтобы понять, как работает этот паттерн, сперва представьте запрос по некоему URL, возвращающий JSON-данные. Программа на JavaScript могла бы запросить этот URL, к

примеру, посредством XMLHttpRequest. Предположим, UserId объекта Foo равен 1234. Браузер, запрашивающий URL `http://server2.example.com/Users/1234`, передав Id равный 1234, получил бы ответ вроде:

```
{"Name": "Foo", "Id": 1234, "Rank": 7}
```

Эти JSON-данные могли бы быть динамически созданными согласно параметрам запроса, переданным в URL.

Ниже HTML-элемент `<script>` указывает в качестве атрибута `src` ссылку, возвращающую JSON:

```
<script type="text/javascript"
        src="http://server2.example.com/Users/1234">
</script>
```

В свою очередь, браузер скачает файл `script`, разберёт его содержимое, интерпретирует сырые JSON-данные как блок и выкинет ошибку синтаксиса. Даже если данные были интерпретированы как литеральный объект JavaScript, к нему невозможно получить доступ из JavaScript, выполняемого в браузере, поскольку без присвоения переменной объектные литералы недоступны.

В паттерне JSONP URL, на который указывает атрибут src тэга `<script>`, возвращает данные JSON, обёрнутые в вызов функции. В подобном случае функция, уже определённая в среде JavaScript, может манипулировать JSON-данными. Начинка JSONP может выглядеть так:

```
functionCall({"Name": "Foo", "Id": 1234, "Rank": 7});
```

Вызов функции это и есть «P» в слове JSONP — «padding» (набивка, «отступ») вокруг чистого JSON. По соглашению, браузер передаёт имя функции обратного вызова как именованный параметр запроса, обычно используя имя `jsonp` или `callback` в запросе к серверу, то есть,

```
<script type="text/javascript"
        src="http://server2.example.com/Users/1234?jsonp=parseResponse">
</script>
```

В данном примере начинка будет такова:

```
parseResponse({"Name": "Foo", "Id": 1234, "Rank": 7});
```

Набивка

В то время как набивка (префикс) является *обычно* именем функции обратного вызова, определённой внутри контекста выполнения в браузере, она может также быть переменной, оператором if, или любым другим оператором JavaScript. Ответ на JSONP-запрос не является JSON и не парсится как JSON; начинка может быть любым выражением на JavaScript, и вовсе не требует, чтобы внутри обязательно был JSON. Но обычно это фрагмент JavaScript, применяющий вызов функции к неким JSON-данным.

Инъекция *элемента script*

JSONP имеет смысл только когда используется с элементом script. Для каждого нового JSONP-запроса браузер должен добавить новый элемент `<script>` или использовать существующий. Первая манипуляция — добавление нового элемента script — осуществляется через динамическое манипулирование DOM, и известна как *инъекция элемента script*. Элемент `<script>` впрыскивается в HTML DOM, с URL желаемой конечной точки JSONP в атрибуте «src».

Эта динамическая *инъекция элемента script* обычно выполняется вспомогательной библиотекой javascript. У jQuery и других фреймворков имеются вспомогательные функции для JSONP.

Динамически вставляемый элемент script для вызовов JSONP выглядит следующим образом:

```
<script type="text/javascript"
        src="http://server2.example.com/Users/1234?jsonp=parseResponse">
</script>
```

После вставки элемента, браузер обрабатывает его и выполняет HTTP GET для src URL, получая содержимое. Затем браузер обрабатывает возвращённую полезную начинку как javascript. Обычно это выполнение функции.

В этом смысле применение JSONP можно охарактеризовать как *разрешить браузерным страницам обойти политику ограничения домена путём вставки элемента script.*

Основные концепции ASP.NET MVC

Модель-представление-контроллер (*Model-View-Controller*, *MVC*) – архитектурный шаблон, состоящий из трёх компонентов (рис. 1):

1. *Модель* представляет данные, с которыми работает приложение. Модель реализовывает логику обработки данных согласно заданным бизнес-правилам и обеспечивает чтение и сохранение данных во внешних хранилищах.
2. *Представление* обеспечивает способ отображения модели.
3. *Контроллер* обрабатывает внешние запросы и координирует изменение модели и актуальность представления.

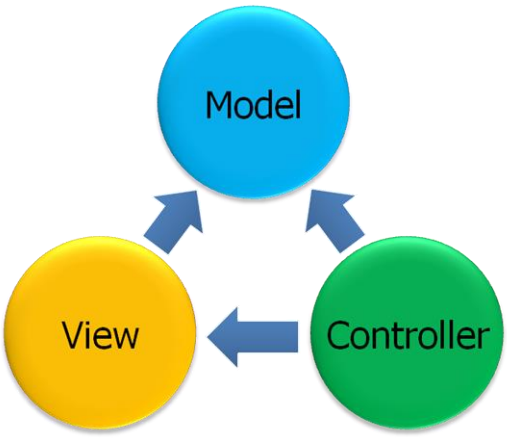


Рис. 1. Модель-представление-контроллер.

Как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Это одно из ключевых достоинств MVC. Оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений и контроллеров для одной модели.

Классический подход к разработке веб-приложений на основе ASP.NET, известный как *ASP.NET Web Forms*, стремится перенести принципы создания обычных оконных приложений в веб-среду. ASP.NET Web Forms предлагает использовать готовые элементы управления и событийную модель программирования с сохранением состояния страницы. Однако такой подход обладает рядом недостатков:

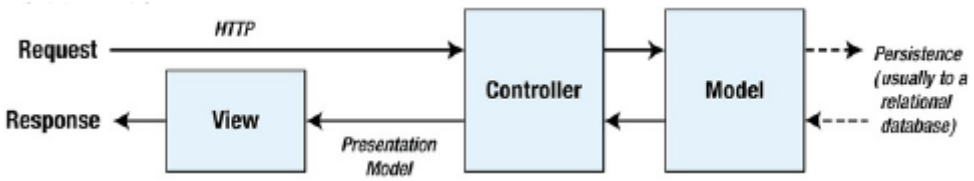
- Сохранение состояния страницы основано на механизме ViewState, что ведёт к передаче больших объёмов данных между клиентом и сервером.
- Понимание этапов жизненного цикла страницы и связанных с ним событий не является тривиальным.
- Разработчик имеет ограниченный контроль над HTML-кодом, так как стандартные элементы управления выполняют свой рендеренг в HTML предопределённым образом.
- Адреса и ссылки, генерируемые ASP.NET, затрудняют индексирование и поисковую оптимизацию сайта.
- Событийная модель провоцирует на использование *антипаттерна Smart UI*, при котором бизнес-правила помещаются не в слой бизнес-логики, а в обработчики событий.
- Модульное тестирование приложений ASP.NET Web Forms затруднено.

ASP.NET MVC – это каркас для разработки веб-приложений, созданный как альтернатива ASP.NET Web Forms. Как следует из названия, ASP.NET MVC базируется на ASP.NET, но предполагает использование шаблона MVC. Ниже перечислены основные отличительные черты ASP.NET MVC.

1. *Открытость*. Исходные коды ASP.NET MVC доступны для анализа.
2. Четкое *разделение компонентов* приложения, опирающееся на использовании шаблона MVC.
3. *Расширяемость*. При работе с ASP.NET MVC практически для каждого элемента каркаса можно использовать либо реализацию по умолчанию, либо наследование или полную замену элемента для удовлетворения специфических нужд.
4. Лучшие возможности для *модульного тестирования*.
5. Полный *контроль* над генерируемой *HTML-разметкой*.
6. *Система маршрутизации* – программист полностью контролирует используемые в приложении URLs и схему их построения.
7. Использование *конвенций именования*: хотя следования данным правилам не обязательно, применение соглашений значительно уменьшает объем кода, создаваемого программистом.
8. Применение возможностей ASP.NET и платформы .NET 3.5. ASP.NET MVC является альтернативой ASP.NET Web Forms, но не отвергает саму платформу ASP.NET. Многие механизмы ASP.NET с успехом применяются в ASP.NET MVC. Возможно построение *гибридных приложений*, сочетающих ASP.NET MVC и ASP.NET Web Forms.

ASP.NET реализация MVC

В MVC контроллеры являются C# классами, как правило, производными от класса System.Web.Mvc.Controller. Каждый открытый (public) метод в классе, производный от Controller, называется методом действия, который связан с настраиваемым URL через систему маршрутизации (роутингом) ASP.NET. При отправке запроса на URL, связанный с методом действия, исполняются выражения в классе контроллера для того, чтобы выполнить некоторые операции над доменной моделью, а затем выбрать представление для отображения клиенту.



ASP.NET MVC Framework обеспечивает поддержку выбора движков представления. Более ранние версии MVC

использовали стандартный ASP.NET движок представления, который обрабатывал ASPX страницы с помощью модернизированной версии синтаксиса разметки Web Forms. MVC 3 ввел движок представления Razor, который был усовершенствован в MVC 4 и который использует полностью другой синтаксис. Visual Studio обеспечивает поддержку IntelliSense для обоих движков представления.

ASP.NET MVC не применяет никаких ограничений на реализацию вашей доменной модели. Вы можете создать модель с помощью обычных C# объектов и осуществлять хранение при помощи любой из баз данных, объектно-реляционных фреймворков или других инструментов хранения данных, поддерживаемых .NET. Visual Studio создает папку /Models в рамках шаблона MVC проекта. Это подходит для простых проектов, но в более сложных приложениях, как правило, доменные модели определяются в отдельный проект Visual Studio.

Что не так с ASP.NET Web Forms?

Со временем использование веб форм в реальных проектах показало некоторые их недостатки:

Вес View State: В результате использования актуального механизма для поддержки состояния между запросами (известного как View State) мы получили большие блоки данных, передаваемые между клиентом и сервером. Эти данные могут достигать сотен килобайт даже для скромных веб приложений, и они идут туда и обратно при каждом запросе, что приводит к увеличению времени отклика и повышению требований к пропускной способности сервера.

Жизненный цикл страницы: Механизм для объединения события со стороны клиента с кодом серверного обработчика события – часть жизненного цикла страницы – может быть чрезвычайно сложным и деликатным. Немногие разработчики добились успеха в манипуляциях с элементами управления во время выполнения кода, не получив ошибок View State или не обнаружив, что некоторые обработчики событий таинственным образом не выполнялись.

Неправильное разделение задач: Модель выделенного кода (code-behind) ASP.NET предоставляет возможность для того, чтобы вынести код приложения за рамки HTML разметки в отдельный класс выделенного кода. Это широко приветствовалось из-за разделения логики и представления, но, в действительности, разработчики вынуждены смешивать код представления (например, манипуляции с деревом серверных элементов управления) с логикой приложения (например, управлением базами данных) в этих же классах выделенного кода, которые становятся просто чудовищными. Конечный результат может быть недолговечным и непонятным.

Ограниченные возможности с HTML: Серверные элементы управления отображают себя как HTML, но не обязательно так, как вы хотите. До версии ASP.NET 4 выходным данным HTML не удавалось соответствовать веб стандартам или хорошо работать с каскадными таблицами стилей (CSS). Также серверные элементы управления генерировали непредсказуемые и сложные значения атрибута ID, к которым трудно получить доступ при помощи JavaScript. Эти проблемы во многом решились в ASP.NET 4 и ASP.NET 4.5, но у вас все еще могут возникнуть сложности в получении того HTML, который вы ожидаете. Абстракции с брешью: Web Forms пытается спрятать HTML и HTTP, где это только возможно. Когда вы пытаетесь реализовать пользовательские механизмы поведения, вы часто можете выпасть из абстракций, которые заставляют вас переделывать механизм обратной передачи событий или выполнять глупые действия, чтобы он сгенерировал желаемый HTML. Кроме того, все эти абстракции могут стать неприятным барьером для компетентных веб разработчиков.

Слабая тестируемость: Разработчики ASP.NET не могли предположить, что автоматизированное тестирование станет важным компонентом разработки программного обеспечения. Не удивительно, что жесткая архитектура, которую они разработали, не подходит для модульного тестирования (юнит-тестирования). С интеграционным тестированием также могут возникнуть проблемы.