**Министерство образования и науки Российской Федерации**

Федеральное государственное автономное образовательное учреждение

высшего образования

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Инженерная школа информационных технологий и робототехники

Направление Информационные системы и технологии

Отделение Информационных технологий

**Описание технологии**

по дисциплине Интернет технологии

Выполнил студент гр. 8И6А \_\_\_\_\_\_\_\_\_\_\_\_ Голушков А.Н.

\_\_\_\_\_\_\_.2019г.

Отчет принят:

Доцент ОИТ \_\_\_\_\_\_\_\_\_\_\_\_ Цапко С.Г.

Томск 2019 г.

Оглавление

[1. Особенности ASP.NET MVC 3](#_Toc28006589)

[2. Контроллеры 4](#_Toc28006590)

[3. Контекст запроса HttpContext. Куки. Сессии 6](#_Toc28006591)

[1.3.1 Отправка ответа 7](#_Toc28006592)

[1.3.2 Сессии 8](#_Toc28006593)

[4. Отправка файлов 8](#_Toc28006594)

[5. Мастер страницы 10](#_Toc28006595)

[1.5.1 ViewStart 12](#_Toc28006596)

[6. Модели и БД 13](#_Toc28006597)

[7. Маршрутизация 15](#_Toc28006598)

[8. Валидация 18](#_Toc28006599)

[9. Защита от межсайтовой подделки 19](#_Toc28006600)

[10. Миграции 20](#_Toc28006601)

[11. Фильтры аутентификации 23](#_Toc28006602)

## Особенности ASP.NET MVC

Платформа ASP.NET MVC представляет собой фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна MVC.

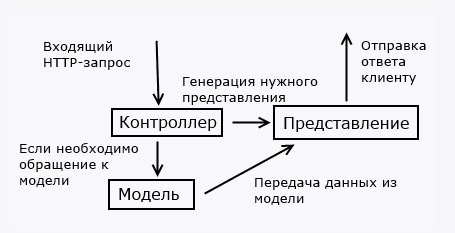
Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:

Контроллер (controller) представляет класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.

Представление (view) - это собственно визуальная часть или пользовательский интерфейс приложения. Как правило, html-страница, которую пользователь видит, зайдя на сайт.

Модель (model) представляет класс, описывающий логику используемых данных.

Общую схему взаимодействия этих компонентов можно представить следующим образом:



Паттерн MVC в ASP.NET

В этой схеме модель является независимым компонентом - любые изменения контроллера или представления не затрагивают модель. Контроллер и представление являются относительно независимыми компонентами, и нередко их можно изменять независимо друг от друга.

Благодаря этому реализуется концепция разделение ответственности, в связи с чем легче построить работу над отдельными компонентами. Кроме того, вследствие этого приложение обладает лучшей тестируемостью. И если нам, допустим, важна визуальная часть или фронтэнд, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на бэкэнде и тестировать контроллер.

Конкретные реализации и определения данного паттерна могут отличаться, но в силу своей гибкости и простоты он стал очень популярным в последнее время, особенно в сфере веб-разработки.

Свою реализацию паттерна представляет платформа ASP.NET MVC. 2013 год ознаменовался выходом новой версии ASP.NET MVC - MVC 5, а также релизом Visual Studio 2013, которая предоставляет инструментарий для работы с MVC5.

## Контроллеры

Контроллер является центральным компонентом в архитектуре MVC. Контроллер получает ввод пользователя, обрабатывает его и посылает обратно результат обработки, например, в виде представления.

При использовании контроллеров существуют некоторые условности. Так, по соглашениям об именовании названия контроллеров должны оканчиваться на суффикс "Controller", остальная же часть до этого суффикса считается именем контроллера.

Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать **адрес\_сайта/Имя\_контроллера/**. Так, по запросу **адрес\_сайта/Home/** система маршрутизации по умолчанию вызовет метод Index контроллера HomeController для обработки входящего запроса. Если мы хотим отправить запрос к конкретному методу контроллера, то нужно указывать этот метод явно: **адрес\_сайта/Имя\_контроллера/Метод\_контроллера**, например, *адрес\_сайта/Home/Buy* - обращение к методу Buy контроллера HomeController.

Контроллер представляет обычный класс, который наследуется от базового класса **System.Web.Mvc.Controller**. В свою очередь класс Controller реализует абстрактный базовый класс ControllerBase, а через него и интерфейс **IController**. Таким образом, формально, чтобы создать свой класс контроллера, достаточно создать класс, реализующий интерфейс IController и имеющий в имени суффикс *Controller*.

Интерфейс IController определяет один единственный метод Execute, который отвечает за обработку контекста запроса:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IController  {      void Execute(RequestContext requestContext);  } |

Теперь создадим какой-нибудь простенький контроллер, реализующий данный интерфейс. В качестве проекта мы можем взять проект из предыдущий главы. Итак, добавим в папку Controllers проекта новый класс (именно класс, а не контроллер) со следующим содержанием:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System.Web.Mvc;  using System.Web.Routing;    namespace BookStore.Controllers  {      public class MyController : IController      {          public void Execute(RequestContext requestContext)          {              string ip = requestContext.HttpContext.Request.UserHostAddress;              var response = requestContext.HttpContext.Response;              response.Write("<h2>Ваш IP-адрес: " + ip + "</h2>");          }      }  } |

При обращении к любому контроллеру система передает в него контекст запроса. В этот контекст запроса включается все: куки, отправленные данные форм, строки запроса, идентификационные данные пользователя и т.д. Реализация интерфейса IController позволяет получить этот контекст запроса в методе Execute через параметр **RequestContext**. В нашем случае мы получаем IP-адрес пользователя через свойство requestContext.HttpContext.Request.UserHostAddress.

Кроме того, мы можем отправить пользователю ответ с помощью объекта **Response** и его метода Write.

Таким образом, перейдя по пути **адрес\_сайта/My/**, пользователь увидит свой ip-адрес.

Хотя с помощью реализации интерфейса IController очень просто создавать контроллеры, но в реальности чаще оперируют более высокоуровневыми классами, как например класс Controller, поскольку он предоставляет более мощные средства для обработки запросов. И если при реализации интерфейса IController мы имеем дело с одним методом Execute, и все запросы к этому контроллеру, будут обрабатываться только одним методом, то при наследовании класса Controller мы можем создавать множество методов действий, которые будут отвечать за обработку входящих запросов, и возвращать различные результаты действий.

Чтобы создать стандартный контроллер, мы можем также добавить в папку Controllers простой класс и унаследовать от класса Controller, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using System.Web.Mvc;    namespace BookStore.Controllers  {      public class BookShopController : Controller      {          public ActionResult Index()          {              return View();          }      }  } |

## Контекст запроса HttpContext. Куки. Сессии

Нередко для обработки запроса требуется информация о контексте запроса: какой у пользователя браузер, ip-адрес, с какой страницы или сайта пользователь попал к нам. И ASP.NET MVC позволяет получить всю эту информацию, используя объект **HttpContext**.

Хотя в контроллере мы также можем обратиться к объекту ControllerContext, который имеет свойство HttpContext и по сути предоставляет доступ к той же функциональности и информации. Но в то же время между ними есть некоторые различия. Объект HttpContext описывает данные конкретного http-запроса, который обрабатывается приложением. А ControllerContext описывает данные http-запроса непосредственно по отношению к данному контроллеру.

Вся информация о контексте запроса доступна нам через свойства объекта HttpContext. Так, все данные запроса содержится в свойстве Request. **HttpContext.Request** представляет объект класса, унаследованного от HttpRequestBase, и поэтому содержит все его свойства. Рассмотрим некоторые из них:

Получение браузера пользователя: **HttpContext.Request.Browser**

Иногда просто браузера недостаточно, тогда можно обратиться к агенту пользователя: **HttpContext.Request.UserAgent**

Получение url запроса: **HttpContext.Request.RawUrl**

Получение IP-адреса пользователя: **HttpContext.Request.UserHostAddress**

Получение реферера: HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri Так как реферер может быть не определен, то сначала смотрим, не равен ли он null

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public string Index()  {      string browser = HttpContext.Request.Browser.Browser;      string user\_agent = HttpContext.Request.UserAgent;      string url = HttpContext.Request.RawUrl;      string ip = HttpContext.Request.UserHostAddress;      string referrer = HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri;      return "<p>Browser: " + browser+"</p><p>User-Agent: "+user\_agent+"</p><p>Url запроса: "+url+          "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: "+ip+"</p>";  } |

### Отправка ответа

Если HttpContext.Request содержит информацию о запросе, то свойство HttpContext.Response управляет ответом. Оно представляет объект HttpResponse, который передает на сторону клиента некоторые значения: куки, служебные заголовки, сам ответ в виде кода html. Например, установим кодировку ответа: HttpContext.Response.Charset = "iso-8859-2";

Методы объекта HttpResponse позволяют управлять ответом. Например, метод AddHeader позволяет добавить к ответу дополнительный заголовок.

Кроме того, нам необязательно вызывать метод View в действия контроллера, чтобы отправить клиенту ответ запроса. Мы вполне можем воспользоваться методом HttpContext.Response.Write:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public string ContextData()  {      HttpContext.Response.Write("<h1>Hello World</h1>");        string user\_agent = HttpContext.Request.UserAgent;      string url = HttpContext.Request.RawUrl;      string ip = HttpContext.Request.UserHostAddress;      string referrer = HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri;      return "<p>User-Agent: "+user\_agent+"</p><p>Url запроса: "+url+          "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: "+ip+"</p>";  } |

Либо так:

|  |  |
| --- | --- |
| 1  2  3  4 | public void ContextData()  {      HttpContext.Response.Write("<h1>Hello World</h1>");  } |

Метод HttpContext.Response.Write здесь добавляет в поток определенное содержимое, переданное в качестве параметра. Но в реальности, конечно, проще использовать методы, генерирующие объекты ActionResult, например, представления.

Определение пользователя

Также объект HttpContext содержит информацию о пользователе в свойстве **HttpContext.User**:

|  |  |
| --- | --- |
| 1  2  3 | bool IsAdmin = HttpContext.User.IsInRole("admin"); // определяем, принадлежит ли пользователь к администраторам  bool IsAuth=HttpContext.User.Identity.IsAuthenticated; // аутентифицирован ли пользователь  string login = HttpContext.User.Identity.Name; // логин авторизованного пользователя |

Подробнее об идентификации пользователя можно узнать в главе про аутентификацию и авторизацию.

Работа с куки

Чтобы получить куки, нам надо воспользоваться свойством HttpContext.Request.Cookies:

|  |  |
| --- | --- |
| 1 | string id =HttpContext.Request.Cookies["id"].Value; |

В данном случае, если у нас установлена на стороне клиента куки "id", то мы получим ее значение.

Однако прежде чем получать значения куки, их естественно надо установить. Для установки значения куки мы можем использовать свойство HttpContext.Response. Например, установим в куки значение "id":

|  |  |
| --- | --- |
| 1 | HttpContext.Response.Cookies["id"].Value = "ca-4353w"; |

### Сессии

Сессии также, как и куки, используются для хранения некоторых данных, которые можно получить в любом месте приложения. Для работы с ними используется объект **Session**. Например, установим в одном методе контроллера мы можем установить значение сессии:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult Index()  {      Session["name"] = "Tom";      return View();  } |

А получить можно в другом методе:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public string GetName()  {      var val = Session["name"];      return val.ToString();  } |

Если мы хотим удалить значение сессии для ключа name, мы можем просто присвоить значение null: Session["name"]=null;

## Отправка файлов

Для отправки клиенту файлов предназначен **FileResult**. Однако так как это абстрактный класс, то фактически мы будем иметь дело с его наследниками:

**FileContentResult**: отправляет клиенту массив байтов, считанный из файла

**FilePathResult**: представляет простую отправку файла напрямую с сервера

**FileStreamResult**: данный класс создает поток - объект System.IO.Stream, с помощью которого считывает и отправляет файл клиенту

Во всех трех случаях для отправки файлов применяется метод **File**, который и возвращает объект FileResult. Только в зависимости от выбранного способа используется соответствующая перегруженная версия этого метода.

Чтобы отправить файл из файловой системы (то есть использование объекта FilePathResult), нам надо указать в методе File три параметра: путь к файлу на стороне сервера, тип содержимого и имя файла для принимающей стороны (имя файла необязательно, и можно обойтись в принципе только двумя параметрами).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public FileResult GetFile()  {      // Путь к файлу      string file\_path = Server.MapPath("~/Files/PDFIcon.pdf");      // Тип файла - content-type      string file\_type="application/pdf";      // Имя файла - необязательно      string file\_name = "PDFIcon.pdf";      return File(file\_path,file\_type,file\_name);  } |

Предполагается, что у нас в проекте есть папка Files, в которой лежит файл PDFIcon.pdf. Метод **Server.MapPath** позволяет построить полный путь к ресурсу из каталога в проекте. Но также можно использовать и абсолютные пути, обращаясь к любому файлу в файловой системе, например, string file\_path = @"C:\Book\PDFIcon.pdf";

И, при обращении, например, по пути *Home/GetFile* нам будет предложено сохранить данный файл на локальном компьютере.

Похожим образом работает и классы FileContentResult, только вместо имени файла в методе File указывается массив байтов, в который был считан файл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Отправка массива байтов  public FileResult GetBytes()  {      string path = Server.MapPath("~/Files/PDFIcon.pdf");      byte[] mas = System.IO.File.ReadAllBytes(path);      string file\_type = "application/pdf";      string file\_name = "PDFIcon.pdf";      return File(mas, file\_type, file\_name);  } |

И если мы хотим возвратить объект FileStreamResult, то в качестве первого аргумента в методе File идет объект Stream для отправляемого файла:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Отправка потока  public FileResult GetStream()  {      string path = Server.MapPath("~/Files/PDFIcon.pdf");      // Объект Stream      FileStream fs = new FileStream(path, FileMode.Open);      string file\_type = "application/pdf";      string file\_name = "PDFIcon.pdf";      return File(fs, file\_type, file\_name);  } |

## Мастер страницы

Мастер-страницы используются для создания единообразного, унифицированного вида сайта. По сути мастер-страницы - это те же самые представления, но позволяющие включать в себя другие представления. Например, можно определить на мастер-странице общие для всех остальных представлений элементы, а также подключить общие стили и скрипты. В итог нам не придется на каждом отдельном представлении прописывать путь к файлам стилей, а потом при необходимости его изменять. А используемые на мастер-страницах заполнители или плейсхолдеры позволят вставить на их место другие представления.

По умолчанию при создании нового проекта ASP.NET MVC 5 в проект уже добавляется мастер-страница под названием *\_Layout.chtml*, которую можно найти в каталоге *Views/Shared*. В приложении из второй главы мы ее изменили следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>@ViewBag.Title</title>      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />  </head>    <body>      <nav>          <ul class="menu">              <li>@Html.ActionLink("Главная", "Index", "Home")</li>          </ul>      </nav>      @RenderBody()  </body>  </html> |

На первый взгляд это обычное представление за одним исключением: здесь используется метод @RenderBody(), который является заместителем и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу. В итоге мы сможем легко установить для представлений веб-приложения единообразный стиль.

Чтобы в представлении использовать мастер-страницу, нам надо в секции **Layout** указать путь к нужной мастер-странице. Например, в представлении Index.cshtml можно определить мастер-страницу так:

|  |  |
| --- | --- |
| 1  2  3 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  } |

Если мы не используем мастер-страницу, то указываем Layout = null;.

Также необязательно для всех представлений использовать одну и ту же мастер-страницу. Можно определить несколько мастер-страниц, например, одну для форума, одну для блога и т.д., и в зависимости от раздела сайта подключать нужную. Добавляются в проект они также как и обычные представления.

Мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое. Например, добавим к мастер-странице секцию footer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>@ViewBag.Title</title>      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />  </head>    <body>      <nav>          <ul class="menu">              <li>@Html.ActionLink("Главная", "Index", "Home")</li>          </ul>      </nav>      @RenderBody()      <footer>@RenderSection("Footer")</footer>  </body>  </html> |

Теперь при запуске предыдущего представления Index мы получим ошибку, так как секция Footer не определена. По умолчанию представление должно передавать содержание для каждой секции мастер-страницы. Поэтому добавим вниз представления Index секцию footer. Это мы можем сделать с помощью выражения @section:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }  <!-- здесь остальное содержание -->  @section Footer {      Все права защищены. Syte Corp. 2012.  } |

Но при таком подходе, если у нас есть куча представлений, и мы вдруг захотели определить новую секцию на мастер-странице, нам придется изменить все имеющиеся представления, что не очень удобно. В этом случае мы можем воспользоваться одним из вариантов гибкой настройки секций.

Первый вариант заключается в использовании перегруженной версии метода **RenderSection**, которая позволяет указать, что данную секцию не обязательно определять в представлении. Чтобы отметить секцию Footer в качестве необязательной, надо передать в метод в качестве второго параметра значение false:

|  |  |
| --- | --- |
| 1 | <footer>@RenderSection("Footer", false)</footer> |

Второй вариант позволяет задать содержание секции по умолчанию, если данная секция не определена в представлении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <footer>      @if (IsSectionDefined("Footer")) {          @RenderSection("Footer")      }      else      {          <span>Содержание элемента footer по умолчанию.</span>      }  </footer> |

### ViewStart

Если у нас в проекте пара-тройка представлений, мы легко можем изменить вручную для каждого описание мастер-страницы в секции Layout, если, например, мы решим использовать другую мастер-страницу. Но если у нас много представлений, то это делать будет не очень удобно.

Для более гибкой настройки представлений предназначена страница \_ViewStart.cshtml. Код этой страницы выполняется до кода любого из представлений, расположенных в том же каталоге. Данный файл последовательно применяется к каждому представлению, находящемуся в одном каталоге.

При создании проекта ASP.NET MVC 5 в каталог Views уже по умолчанию добавляется файл \_ViewStart.cshtml. Этот файл определяет мастер-страницу, используемую по умолчанию:

|  |  |
| --- | --- |
| 1  2  3 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  } |

Данный код выполняется до любого другого кода, определенного в представлении, поэтому из других представлений мы можем удалить секцию Layout. Если же представление должно использовать другую мастер-страницу, то мы просто переопределяем свойство Layout, дописывая его определение в начало представления.

## Модели и БД

Все сущности в приложении принято выделять в отдельные модели. В зависимости от поставленной задачи и сложности приложения можно выделить различное количество моделей. Так, в тестовом приложении из второй главы использовались две модели - класс для книги и класс для покупки книги.

Модели представляют собой простые классы и располагаются в проекте в каталоге *Models*. Модели описывают логику данных. Например, модель представляющая книгу и ее покупку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Book  {      // ID книги      public int Id { get; set; }      // название книги      public string Name { get; set; }      // автор книги      public string Author { get; set; }      // цена      public int Price { get; set; }  }  public class Purchase  {      public int PurchaseId { get; set; }      public string Person { get; set; }      public string Address { get; set; }      public int BookId { get; set; }      public DateTime Date { get; set; }  } |

Модель необязательно состоит только из свойств, кроме того, она может иметь конструктор, какие-нибудь вспомогательные методы. Но главное не перегружать класс модели и помнить, что его предназначение - описывать данные. Манипуляции с данными и бизнес-логика - это больше сфера контроллера.

Данные моделей, как правило, хранятся в базе данных. Для работы с базой данных очень удобно пользоваться фреймворком **Entity Framework**, который позволяет абстрагироваться от написания sql-запросов, от строения базы данных и полностью сосредоточиться на логике приложения.

Если при создании проекта MVC 5 вы выберете в качестве типа аутентификации "No Authentication", то после создания проекта в него надо будет подключить EntityFramework через пакетный менеджер NuGet, как описывалось во второй главе.

В качестве альтернативы NuGet можно использовать консоль пакетного менеджера. Для этого в меню Visual Studio выберем View -> Other Windows -> Package Manager Console. После этого внизу студии откроется консоль пакетного менеджера. В ней введем такую команду:

PM> Install-Package EntityFramework -Version 6.0.2

После ввода команды будет загружен и установлен пакет Entity Framework. Иногда этой консолью предпочтительнее пользоваться при установке пакетов, чем менеджером NuGet, так как менеджер NuGet может немного опаздывать за выпуском последних версий пакетов. Либо наоборот, нам надо установить пакеты более ранней версии, а NuGet может предложить только текущую версию.

Entity Framework поддерживает подход "Code first", который предполагает сохранение или извлечение информации из БД на SQL Server без создания схемы базы данных или использования дизайнера в Visual Studo. Наоборот, мы создаем обычные классы, а Entity Framework уже сам определяет, как и где сохранять объекты этих классов.

Для подключения к базе данных через Entity Framework, нам нужен посредник - **контекст данных**. Контекст данных представляет собой класс, производный от класса **DbContext**. Контекст данных содержит одно или несколько свойств типа DbSet<T>, где T представляет тип объекта, хранящегося в базе данных. Например, создадим контекст данных для работы с вышеприведенными моделями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using System;  using System.Collections.Generic;  using System.Web;  using System.Data.Entity;    namespace BookStore.Models  {      public class BookContext : DbContext      {          public DbSet<Book> Books { get; set; }          public DbSet<Purchase> Purchases { get; set; }      }  } |

С помощью свойств Books и Purchases мы получаем доступ к данным соответствующих моделей, которые хранятся в базе данных.

## Маршрутизация

В предыдущих главах мы так или иначе сталкивались с маршрутизацией. Например, при обращении к некоторому действию контроллера мы набирали в адресной строке браузера *http://localhost:3456/Home/Index*, где *Home* - имя контроллера без префикса *Controller*, а *Index* - имя метода действия этого контроллера. Если метод Index принимал бы какой-нибудь параметр, например, типа int: public ActionResult Index(int Id), то мы могли обратиться к этому методу и передать значение в его параметр с помощью следующей строки: *http://localhost:3456/Home/Index/5*. В этом плане все очень просто. Но мы не говорили еще о том, почему мы должны прописывать маршрут именно так, и как мы собственно можем управлять маршрутами.

Рассмотрим механизм определения маршрутов. В ASP.NET MVC 5 все определения маршрутов находятся в файле **RouteConfig.cs**, который располагается в проекте в папке *App\_Start*. Если мы на него посмотрим, то увидим настройки маршрута по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using System.Web.Routing;    namespace Mvc5RoutesApp  {      public class RouteConfig      {          public static void RegisterRoutes(RouteCollection routes)          {              routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");                routes.MapRoute(                  name: "Default",                  url: "{controller}/{action}/{id}",                  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }              );          }      }  } |

Установкой маршрутов занимается статический метод RegisterRoutes. Однако на этом определение маршрутов не заканчивается, так как нам еще надо вызвать этот метод в приложении при запуске. Это производится в файле Global.asax в методе **Application\_Start**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | protected void Application\_Start()  {      AreaRegistration.RegisterAllAreas();      FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);      RouteConfig.RegisterRoutes(RouteTable.Routes);      BundleConfig.RegisterBundles(BundleTable.Bundles);  } |

Вызов метода RouteConfig.RegisterRoutes(RouteTable.Routes); как раз и производит регистрацию маршрутов в приложении.

Теперь разберем определение маршрута по умолчанию в файле **RouteConfig.cs**. Первая строка routes.IgnoreRoute("{resource}.axd/{\*pathInfo}"); отключает обработку запросов для некоторых файлов, например с расширением \*.axd (WebResource.axd).

Далее идет собственно определение маршрута по умолчанию. Метод routes.MapRoute выполняет сопоставление маршрута запросу. В перегруженных версиях данного метода мы можем задать дополнительные параметры сопоставления.

Разберем параметры метода. Вначале с помощью свойства name задается имя маршрута - *Default*. Второй параметр - url задает шаблон строки запроса или **шаблон Url**, с которым будет сопоставляться данный маршрут.

Шаблон URL включает в себя несколько сегментов, которые заключены в фигурные скобки. В данном случае сегмент представляет часть запроса, которая находится между слешами. Каждый такой сегмент шаблона содержит параметр. Эти параметры называются параметрами URL. В данном случае это параметры controller, action и id. Но вообще параметры могут иметь различные имена, включающие любые алфавитно-цифровые символы.

При получении запроса механизм маршрутизации парсит строку URL и помещает значения маршрута в словарь - в объект RouteValueDictionary, доступный через контекст приложения RequestContext. В качестве ключей в нем применяются имена параметров URL, а соответствующие сегменты URL выступают в качестве значений. Например, у нас есть следующий URL запроса: *http://localhost/Home/Index/5*, то в этом случае образуются следующие пары ключей и значений в словаре RouteValueDictionary:

|  |  |
| --- | --- |
| **Параметр** | **Значение** |
| controller | Home |
| action | Index |
| id | 5 |

Третий параметр метода routes.MapRoute - defaults определяет значения по умолчанию для маршрута. Например, если вдруг в строке запрос указаны не все параметры, а сам запрос, к примеру, идет по адресу *http://localhost/*, то система маршрутизации вызовет метод Index контроллера Home, как указано в параметре defaults. Также, если мы не укажем метод контроллера, например, *http://localhost/Home/*, также будет вызван метод Index контроллера Home.

Поэтому если мы захотим, к примеру, чтобы у нас по умолчанию клиент обращался не к методу Index контроллера HomeController, а, например, к методу Show контроллера BookController, то мы можем соответственно изменить значения данного параметра:

|  |  |
| --- | --- |
| 1 | defaults: new { controller = "Book", action = "Show", id = UrlParameter.Optional } |

Последний параметр объявлен как необязательный id = UrlParameter.Optional, поэтому, если он не указан в строке запроса, он не будет учитываться и передаваться в словарь параметров RouteValueDictionary. Например, запрос *http://localhost/Home/Create/3* вызовет метод Create контроллера HomeController, передав в этот метод в качестве параметра число 3. В то же время запрос *http://localhost/Home/Create/* также вызовет метод Create контроллера HomeController, хотя последний параметр в нем не указан.

Таким образом, настройки по умолчанию позволяют нам не указывать в строке запроса полностью название контроллера и его метода. Но в случае, если такие настройки не заданы, мы должны определять в строке запроса контроллер и его метод. Например, изменим установку маршрутов в файле **RouteConfig.cs** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class RouteConfig  {      public static void RegisterRoutes(RouteCollection routes)      {          routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");            routes.MapRoute(                  name: "Default",                  url: "{controller}/{action}"          );      }  } |

Теперь обратимся к нашему приложению, набрав в строке браузера только адрес сайта, например, *http://mysyte.com/*. В этом случае мы получим информацию об ошибке. Ошибка будет состоять в том, что теперь нам надо полностью набирать в строке запроса адрес ресурса. Поэтому следующий адрес *http://mysyte.com/Home/Index* будет нормально работать (если, конечно, в приложении определен контроллер Home с методом Index и соответствующим ему представлением).

И если мы теперь перейдем по адресу *http://localhost/Home/*, как мы это делали выше, то опять получим ошибку, так как в строке запроса указан только один сегмент. А в определении маршрута у нас указано два сегмента - *{controller}/{action}*. Если для параметров не определены значения по умолчанию, то строка запроса должна иметь такое же число сегментов, для которых не определены значения по умолчанию.

В то же время если запрос будет состоять из трех сегментов, например, *http://localhost/Home/Index/1*, то мы также получим ошибку, потому что число сегментов в запросе больше числа, определенного в шаблоне URL данного маршрута.

## Валидация

Кроме валидации на стороне клиента, мы можем осуществлять валидацию и внутри контроллера. Делается это с помощью проверки значения свойства **ModelState.IsValid**.

Объект ModelState сохраняет все значения, которые пользователь ввел для свойств модели, а также все ошибки, связанные с каждым свойством и с моделью в целом. Если в объекте ModelState имеются какие-нибудь ошибки, то свойство ModelState.IsValid возвратит false:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | [HttpPost]  public ActionResult Create(Book book)  {      if (ModelState.IsValid)      {          ViewBag.Message = "Valid";          db.Books.Add(book);          db.SaveChanges();          return RedirectToAction("Index");      }      ViewBag.Message = "Non Valid";      return View(book);  } |

Допустим, значение для свойства Name у нас не указано, и мы отсылаем значения модели на сервер. Поскольку к свойству применяется атрибут Required, то после привязки модели при валидации мы получим следующие значения:

|  |  |
| --- | --- |
| 1  2  3 | ModelState.IsValid == false  ModelState.IsValidField("Name") == false  ModelState["Name"].Errors.Count > 0 |

Мы также можем проверять корректность значений отдельных свойств модели:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | [HttpPost]  public ActionResult Create(Book book)  {      if (string.IsNullOrEmpty(book.Name))      {          ModelState.AddModelError("Name", "Некорректное название книги");      }      else if (book.Name.Length > 5)      {          ModelState.AddModelError("Name", "Недопустимая длина строки");      }        if (ModelState.IsValid)      {          ViewBag.Message = "Валидация пройдена";          db.Books.Add(book);          db.SaveChanges();          return RedirectToAction("Index");      }        ViewBag.Message = "Запрос не прошел валидацию";      return View(book);  } |

В данном случае метод ModelState.AddModelError добавляет для свойства, указанного в качестве первого параметра (в данном случае Name) ошибку "Недопустимая длина строки". При использовании хелперов впоследствии мы можем вывести данное сообщение об ошибке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div class="editor-label">      @Html.LabelFor(model => model.Name)  </div>  <div class="editor-field">      @Html.EditorFor(model => model.Name)      @Html.ValidationMessageFor(model => model.Name)  </div> |

## Защита от межсайтовой подделки

Наиболее распространенным подходом к защите от атак CSRF является использование *шаблона токена синхронизатора* (STP). STP используется, когда пользователь запрашивает страницу с данными формы:

Сервер отправляет клиенту маркер, связанный с удостоверением текущего пользователя.

Клиент отправляет на сервер токен, чтобы выполнить проверку.

Если сервер получает маркер, который не соответствует удостоверению аутентифицированного пользователя, запрос отклоняется.

Маркер является уникальным и непредсказуемым. Маркер также можно использовать для обеспечения надлежащей последовательности последовательности запросов (например, для обеспечения последовательности запроса: страница 1 – стр. 2 – стр. 3). Все формы в ASP.NET Core шаблонах MVC и Razor Pages создают маркеры защиты от подделки. В следующей паре примеров представления создаются маркеры защиты от подделки:

<form asp-controller="Manage" asp-action="ChangePassword" method="post">

...

</form>

@using (Html.BeginForm("ChangePassword", "Manage"))

{

...

}

Явно добавьте токен защиты от подделки к элементу <form> без использования вспомогательных функций тегов при помощи [@Html.AntiForgeryToken](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.htmlhelper.antiforgerytoken)поддержки HTML:

<form action="/" method="post">

@Html.AntiForgeryToken()

</form>

В каждом из предыдущих случаев ASP.NET Core добавляет скрытое поле формы, аналогичное следующему:

<input name="\_\_RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ... s2-m9Yw">

## Миграции

Нередко возникает ситуация, когда модель меняется. Например, мы решили внести в нее новые свойства. Но при этом у нас уже имеется существующая база данных, в которой есть какие-то данные. И чтобы без потерь обновить базу данных ASP.NET MVC предлагает нам такой механизм как миграции. Например, у нас есть простая модель User:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class User  {      public int Id { get; set; }      public string Name { get; set; }  } |

Соответсвенно есть контекст данных, через который мы работаем с бд:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class UserContext : DbContext  {      public UserContext() :              base("DefaultConnection")      { }      public DbSet<User> Users { get; set; }  } |

И допустим, у нас есть вся инфраструктура для работы с этой моделью - представления, контроллеры, и у нас есть уже в базе данных несколько объектов данной модели. Но в какой-то момент мы решили изменить модельную базу приложения. Например, мы добавили еще одно поле в модель User:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class User  {      public int Id { get; set; }      public string Name { get; set; }      public int Age { get; set; }  } |

Кроме того, мы решили добавить еще одну модель, например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Company  {      public int Id { get; set; }      public string Name { get; set; }  } |

Таким образом, контекст данных у нас уже меняется следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class UserContext : DbContext  {      public UserContext() :              base("DefaultConnection")      { }      public DbSet<User> Users { get; set; }      public DbSet<Company> Companies { get; set; }  } |

Мы можем добавить в представления для модели User дополнительное поле для свойства Age, можем создать контроллер и представления для новой модели, но при попытке добавить новый объект в бд, мы будем получать ошибку:

Контекст данных изменился, и теперь нам надо провести миграцию от старой схемы базы данных к новой. И первым делом найдем внизу Visual Studio окно Package Manager Console, введем в нем команду: enable-migrations и нажмем Enter:

После выполнения этой команды Visual Studio в проекте будет создана папка Migrations, в которой можно найти файл *Configuration.cs*. Этот файл содержит объявление одноименного класса Configuration, который устанавливает настройки конфигурации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | namespace MigrationApp.Migrations  {      using System;      using System.Data.Entity;      using System.Data.Entity.Migrations;      using System.Linq;        internal sealed class Configuration : DbMigrationsConfiguration<MigrationApp.Models.UserContext>      {          public Configuration()          {              AutomaticMigrationsEnabled = false;              ContextKey = "MigrationApp.Models.UserContext";          }            protected override void Seed(MigrationApp.Models.UserContext context)          {          }      }  } |

В методе Seed можно проинициализировать базу данных начальными данными. Теперь нам надо создать саму миграцию. Там же в консоли Package Manager Console введем команду:

PM> Add-Migration "MigrateDB"

После этого Visual Studio автоматически сгенерирует класс миграции:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | namespace MigrationApp.Migrations  {      using System;      using System.Data.Entity.Migrations;        public partial class MigrateDB : DbMigration      {          public override void Up()          {              CreateTable(                  "dbo.Companies",                  c => new                      {                          Id = c.Int(nullable: false, identity: true),                          Name = c.String(),                      })                  .PrimaryKey(t => t.Id);                AddColumn("dbo.Users", "Age", c => c.Int(nullable: false));          }            public override void Down()          {              DropColumn("dbo.Users", "Age");              DropTable("dbo.Companies");          }      }  } |

В методе Up с помощью вызова метода CreateTable создается таблица "dbo.Companies" и производится ее настройка: создание столбцов, установка ключей. И также добавляется новый столбец Age в уже имеющуюся таблицу. Метод Down удаляет столбец и таблицу на случай, если они существуют. Фактически эти методы равнозначные выражению ALTER в языке SQL, которое меняет структуру базы данных и ее таблиц.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

PM> Update-Database

После этого, если мы посмотрим на состав базы данных, то увидим, что к ней были применены изменения в соответствии с выполненной миграцией:

Итак, миграция выполнена, и мы можем уже использовать обновленные модели и контекст данных.

## Фильтры аутентификации

Фильтры аутентификации появились в ASP.NET MVC 5. Их предназначение - управление аутентификацией пользователей.

Фильтры аутентификации срабатывают до любого другого фильтра и выполнения метода, а также тогда, когда метод уже завершил выполнение, но его результат - объект ActionResult - не обработан.

Любой фильтр аутентификации реализует интерфейс IAuthenticationFilter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace System.Web.Mvc.Filters  {      public interface IAuthenticationFilter      {          void OnAuthentication(AuthenticationContext filterContext);          void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext);      }  } |

Реализация метода OnAuthentication призвана обеспечить проверку пользователя: аутентифицирован ли он в системе. А метод OnAuthenticationChallenge используется для ограничения доступа для аутентифицированного пользователя.

Посмотрим на примере и создадим свой фильтр аутентификации. Возьмем какой-либо проект MVC 5, который использует аутентификацию. Создадим в нем для хранения фильтра специальную папку Filters и добавим в нее следующий класс *MyAuthAttribute*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using System.Web.Mvc.Filters;    namespace AuthFilterApp.Filters  {      public class MyAuthAttribute : FilterAttribute, IAuthenticationFilter      {          public void OnAuthentication(AuthenticationContext filterContext)          {              var user = filterContext.HttpContext.User;              if (user == null || !user.Identity.IsAuthenticated)              {                  filterContext.Result = new HttpUnauthorizedResult();              }          }            public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)          {              var user = filterContext.HttpContext.User;              if(user==null || !user.Identity.IsAuthenticated)              {                  filterContext.Result = new RedirectToRouteResult(                      new System.Web.Routing.RouteValueDictionary {                      { "controller", "Account" }, { "action", "Login" }                     });              }          }      }  } |

Во-первых, чтобы создать свой фильтр, нам надо унаследовать класс от класса FilterAttribute, а также реализовать интерфейс IAuthenticationFilter. Реализацию двух его методов довольно проста. Каждый метод принимает в качестве параметра filterContext, который унаследован от контекста контроллера ControllerContext и содержит всю необходимую информацию о контексте запроса. В том числе он позволяет узнать всю информацию о пользователе, аутентифицирован ли он и т.д. И если пользователь не аутентифицирован, то мы возвращаем определенный результат: filterContext.Result = new HttpUnauthorizedResult(), который говорит о том, что доступ к данному ресурсу для пользователя запрещен. Во втором методе подобным образом мы перенаправляем пользователя на страницу логина через filterContext.Result = new RedirectToRouteResult(new System.Web.Routing.RouteValueDictionary {{ "controller", "Account" }, { "action", "Login" }});

Простенький фильтр готов, и теперь мы модем его применить. Например, у нас есть по умолчанию в проекте контроллер HomeController. Применим фильтр к одному из его действий:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class HomeController : Controller  {      [MyAuthAttribute]      public ActionResult Index()      {          return View();      }  } |

Теперь, если мы не залогинились на сайте, то доступ к действию будет запрещен, и нас будет перенаправлять на страницу логина.