что за паттерн MVC, ну и вообще рассказать про его составляющие, как связаны

жизненный цикл model, view, controller (контроллер и модель живет ровно 1 запрос, представление в c# файл при первом обращение, умирает при перезагрузке)

1. **MVC:** **Model-View-Controller** – патерн проектирования, в котором приложение состоит из трех взаимодействующих компонентов: модель, представление, контроллер. В правильно разработанном MVC-приложений компоненты *относительно* независимы. Каждый компонент имеет свою зону ответственности: модель – данные и бизнес-логика; представление – динамическое формирование разметки для отправки клиенту; контроллер – обработка запроса, формирование экземпляра модели, вызов **Razor Engine**.
2. **MVC/View:** типичная схема применения form



Концепция паттерна MVC 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.

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**.

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

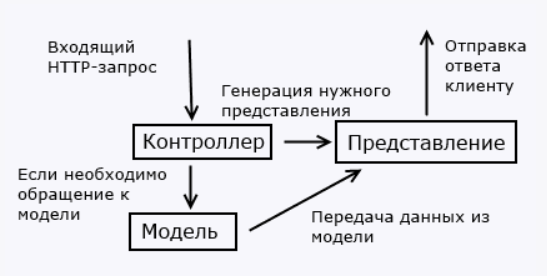
Хотя представление содержит. главным образом, код html, оно не является html-страницей. При компиляции приложения на основе требуемого представления сначала генерируется класс на языке C#, а затем этот класс компилируется.

При вызове метода View контроллер не производит рендеринг представления и не генерирует разметку html. Контроллер только готовит данные и выбирает, какое представление надо возвратить в качестве объекта ViewResult. Затем уже объект ViewResult обращается к движку представления для рендеринга представления в выходной результат.

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

Все сущности в приложении принято выделять в отдельные модели. В зависимости от поставленной задачи и сложности приложения можно выделить разное количество моделей.

Общую схему взаимодействия этих компонентов можно представить следующим образом:



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

Основным элементом в архитектуре ASP.NET Core MVC является **контроллер**. При получении запроса система маршрутизации выбирает для обработки запроса нужный контроллер и передает ему данные запроса. Контроллер обрабатывает эти данные и посылает обратно результат обработки.

Обычно при обращении к веб-приложению пользователь ожидает получить веб-страницу с какими-нибудь данными. В MVC для этого, как правило, используются **представления**, которые определяют внешний вид приложения и на основе которых потом формируется веб-страница. В ASP.NET MVC Core представления - это файлы с расширением cshtml, которые содержат код пользов. интерфейса в основном на языке html.

Одним из ключевых компонентов паттерна MVC являются **модели**. Ключевая задача моделей - описание структуры и логики используемых данных.

Как правило, все используемые сущности в приложении выделяются в отдельные модели, которые и описывают структуру каждой сущности. В зависимости от задач и предметной области мы можем выделить различное количество моделей в приложении..

1. После получения IIS запроса на обработку, при первом обращении к ресурсу, создается объект, который представляет домен приложения, внутри которого обрабатывается запрос. Домены приложений изолируют выполняющиеся приложения друг от друга. Уже внутри домена приложения создается объект класса, который предоставляет доступ к информации о приложении.
2. После создания домена приложения создаются объекты **HttpContext, HttpRequest** и **HttpResponse**.
3. На следующей стадии запускается приложение, которое представляет экземпляр. На этой же стадии происходит начальная инициализация приложения в методе Application\_Start, который находится в файле Global.asax.cs.
4. После этого начинается собственно обработка запроса приложением. И первым шагом здесь является установка маршрута. Запрос перехватывается специальным HTTP-модулем. Этот модуль выбирает маршрут, который соответствует входящему запросу.

Весь набор маршрутов определяется в файле RouteConfig.cs.

А сама регистрация маршрутов происходит в методе Application\_Start строкой RouteConfig.RegisterRoutes(RouteTable.Routes).

1. Если модулю удастся сопоставить запрос с одним из маршрутов, то затем выбирает обработчик маршрутов сопоставленного маршрута.
2. Затем у объекта вызывается метод, используемый для обработки запроса.
3. У обработчика IHttpHandler вызывается метод ProcessRequest для обработки запросов.

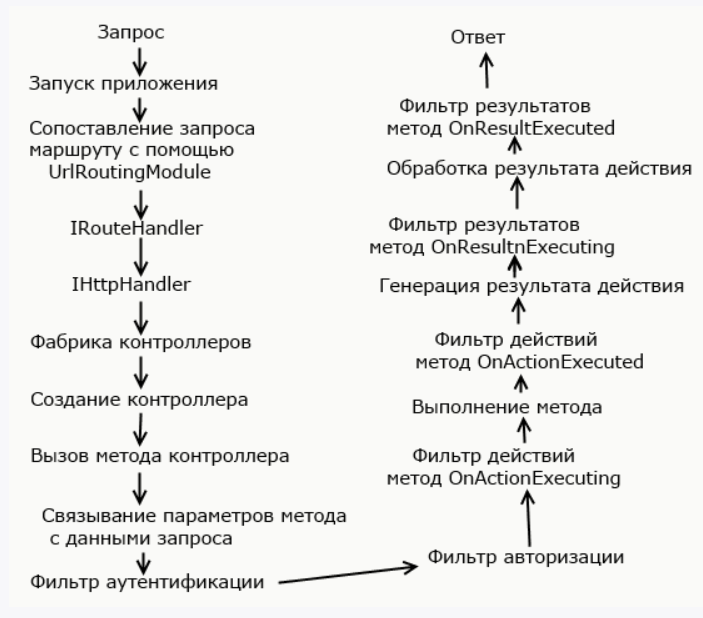
На этом этапе уже происходит непосредственно создание контроллера.

1. После этого начинается собственно выполнение кода контроллера.
2. Далее привязчик модели извлекает данные из запроса, производит их преобразование, форматирование, валидацию и связывает их с параметрами вызываемого метода.
3. Собственно выполнение метода контроллера. Он выполняет определенную логику и на выходе генерирует результат обработки.
4. Генерация результата представляет.

На последнем этапе у каждого объекта вызывается метод, который обрабатывает результат действия. Для объектов происходит отправка результата в выходной поток.

1. И в конце пользователь получает результат обработки своего запроса.

Схематично весь процесс конвейера приложения на ASP.NET MVC 5 можно представить так:



**что такое лэйаут (картинка),**

Когда у нас в проекте много представлений, и все они содержат какие-то общие элементы, то вместо того, чтобы пописывать все эти элементы в каждом представлении, гораздо удобнее задать один общий шаблон. В этом случае при изменении каких-то общих элементов будет достаточно изменить один раз в общем шаблоне, не изменяя всех остальных представлений. В ASP.NET MVC таким шаблоном являются мастер-страницы.

Код мастер-страницы напоминает полноценную веб-страницу: здесь присутсвуют основные теги <html>, <head>, <body> и так далее. И также здесь могут использоваться конструкции Razor. Фактически это то же самое представление. Главное же отличие от обычных представлений состоит в использовании метода **@RenderBody()**, который является плейсхолдером и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу. В итоге мы сможем легко установить для всех представлений веб-приложения единообразный стиль оформления.

При необходимости мы можем использовать несколько мастер-страниц. Добавим в папку Views новую мастер-страницу. Для этого нажмем правой кнопкой мыши на папку Views и в выпадающем меню выберем **Add -> New Item**. Для более быстрого создания мастер-страницы Visual Studio предлагает специальный шаблон **Razor Layout**.

Кроме метода RenderBody(), который вставляет освновное содержимое представлений, мастер-страниц может также использовать специальный метод RenderSection() для вставки секций. Мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое.

Код из *\_ViewStart.cshtml* выполняется до любого кода в представлении. И чтобы переопределить мастер-страницу, в представлении доcтаточно установить свойство Layout. Мы можем не использовать мастер-страницу, тогда нам надо присвоить значение null.

Либо можно использовать какую-нибудь имеющуюся мастер-страницу, указав путь:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @{      ViewData["Title"] = "Home Page";      Layout = "~/Views/\_Master.cshtml";  }  <h2>Представление Index.cshtml</h2> |

**что такое нтмл (на картинке),**

Получает или задает объект htmlHelper, используемый для визуализации элементов html

Хотя мы можем сами написать любой требуемый хелпер, но фреймворк MVC уже предоставляет большой набор встроенных html-хелперов, которые позволяют генерировать ту или иную разметку, главным образом, для работы с формами. Поэтому в большинстве случаев не придется создавать свои хелперы, и можно будет воспользоваться встроенными.

Метод **BeginForm** принимает в качестве параметров имя метода действия и имя контроллера, а также тип запроса. Данный хелпер создает как открывающий, так и закрывающий тег <form>. Поэтому при рендеринге представления нас получится тот же самый html-код, что и с применением тега form. Поэтому оба способа идентичны

**для чего собачка (картинка)**

**@using Имя** То есть мы импортируем пространство имен в представлении

**после собачки С# код, после него ; нельзя ставить**

Стандартное представление очень похоже на обычную веб-страницу с кучей кода html. Однако оно также имеет вставки кода на C#, которые предваряются знаком @.

Здесь важно понимать, что Razor - это не какой-то новый язык, это лишь способ рендеринга представлений, который имеет определенный синтаксис для перехода от разметки html к коду C#.

Использование синтаксиса Razor характеризуется тем, что перед выражением кода стоит знак @, после которого осуществляется переход к коду C#. Существуют два типа переходов: к выражениям кода и к блоку кода.

**Razor Engine. что такое**

Razor - это синтаксис программирования ASP.NET, используемый для создания динамических веб-страниц с языками программирования C # или VB.NET.

Механизм шаблонов, созданный на основе механизма синтаксического анализа Microsoft Razor, RazorEngine позволяет использовать синтаксис Razor для создания динамических шаблонов.

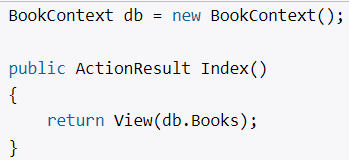
**строго типизированные представления что такое**

строготипизированные представления – представления в которых используется @model/Model (это представление с указанием типа модели в начале файла)

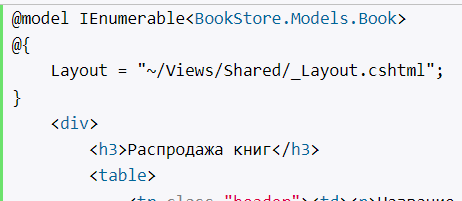
В предыдущих примерах для передачи информации из контроллера в представление использовался объект ViewBag:

Здесь мы получаем доступ к элементам коллекции, заключенной в ViewBag.Books, с помощью переменной с ключевым словом var - то есть тип переменной у нас не задан явно и выводится компилятором. То же самое мы могли бы указать тип модели явно, применив полное имя типа модели.

Хотя примеры с объектом ViewBag работают как надо, но есть и другой способ, иногда более предпочтительный, который заключается в использовании **строго типизированных представлений**. Подобные представления позволяют передавать данные не через объект ViewBag, а напрямую в представление через параметр метода View. Код метода контроллера мог бы выглядеть так:

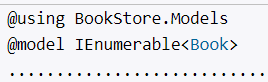


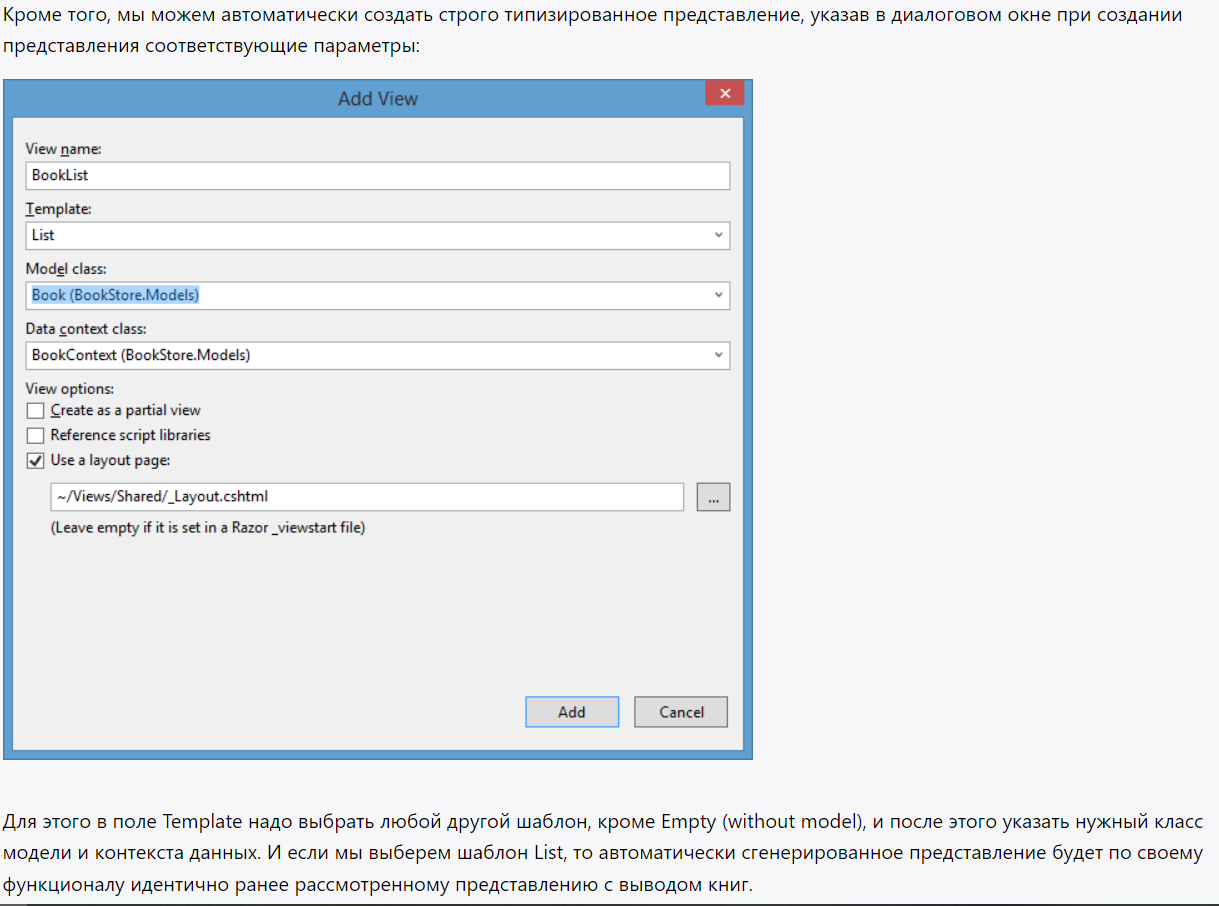
Теперь, чтобы связать представление с передаваемым параметром, надо добавить в представление директиву **@model** с указанием типа передаваемых данных. Поскольку books представляет тип IEnumerable<Book>, то представление будет выглядеть так:

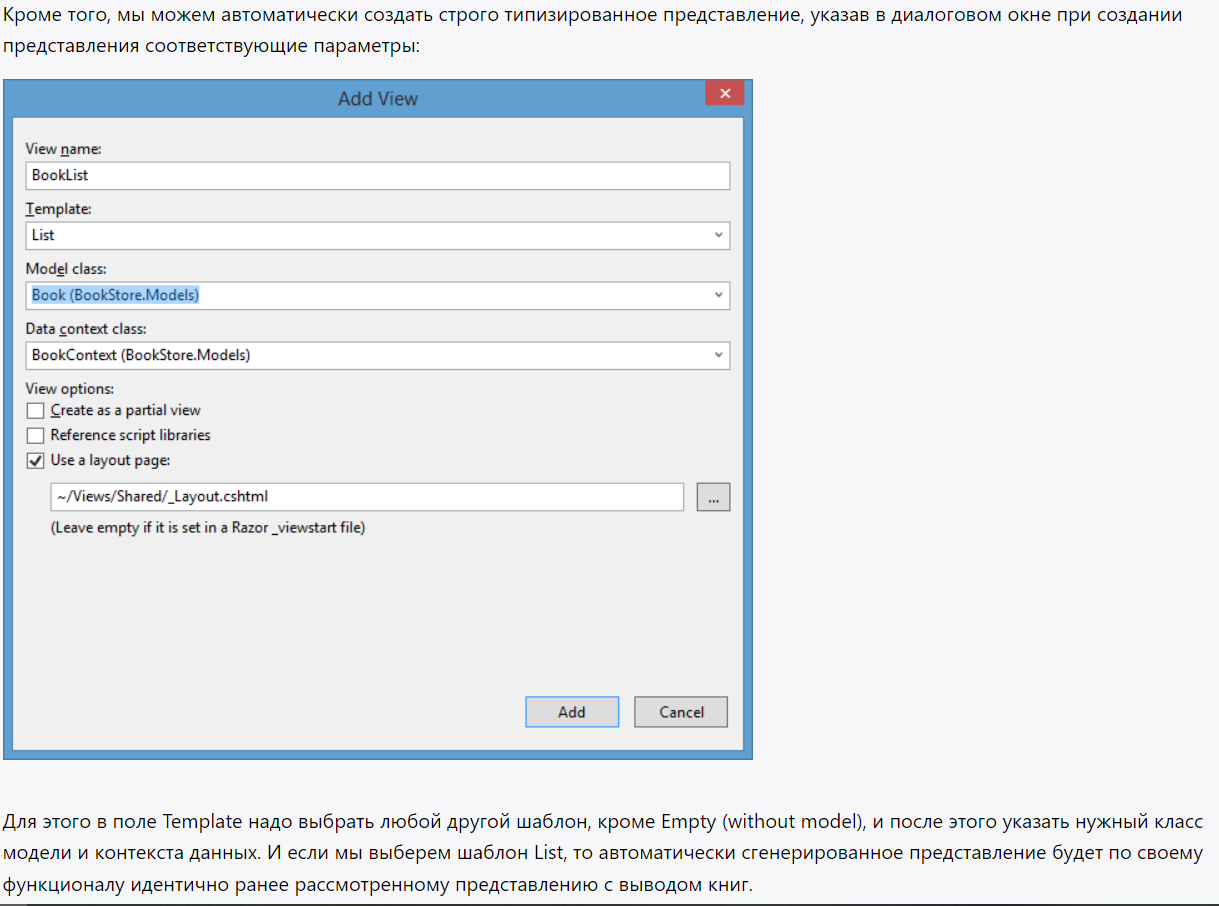


Объект **Model** представляет тип модели, указанной в директиве @model, и будет хранить переданные из контроллера данные.

Но чтобы не писать полностью имя типа модели, мы можем импортировать пространство имен в представлении:



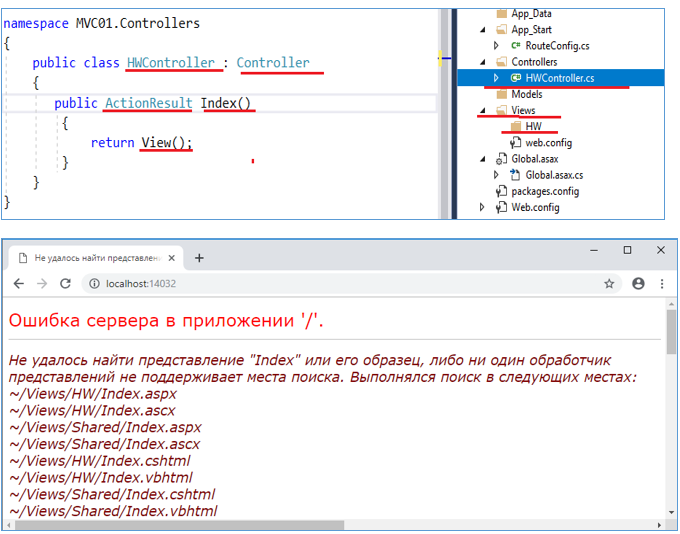




Существует три способа передачи сведений от контроллера в представление в ASP.NET MVC 3.

1. Как строго типизированный объект модели.
2. Как динамический тип (с использованием @model динамического)
3. Использование ViewBag (еще есть viewData, посмотреть шо это)

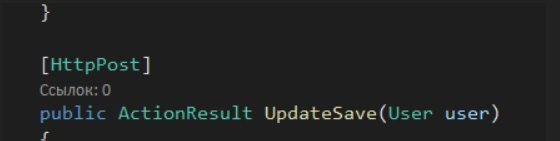
**что будет если ретурн вью оставить пустым**

****

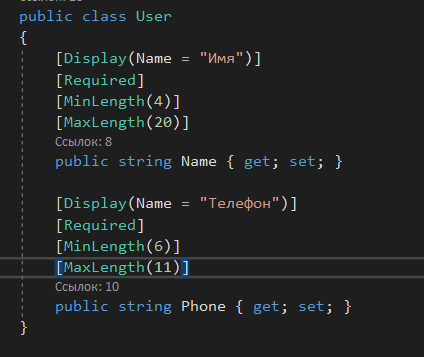
Чтобы произвести рендеринг представления в выходной поток, используется метод View(). Если в этот метод не передается имени представления, то по умолчанию приложение будет работать с тем представлением, имя которого совпадает с именем метода действия.

**что такое аштитипиПОСТ**

Представляет атрибут, используемый для ограничения метода действия таким образом, чтобы этот метод обрабатывал только http-запросы POST



1. Html.EditorForModel, метаданные



Как видно, заголовки у свойств точно такие же, как и названия свойств, что не очень удобно. Ведь мы могли назвать свойства как угодно. К тому же мы не хотим, чтобы названия на русскоязычном сайте были на английском языке.

И тут нам на помощь приходят аннотации данных, а именно атрибут **Display**.

Свойство Name атрибута Display содержит строку, которая будет отображаться вместо имени свойства. Больше нам ничего не надо менять.

.